DOCUMENT RESUME

ED 136 826 IR 004 625

AUTHOR Koffman, Elliot B.; Perry, James
TITLE An Intelligent CAI Monitor and Generative Tutor. Final Report.
INSTITUTION Connecticut Univ., Storrs. Dept. of Electrical Engineering.
SPONS AGENCY National Inst. of Education (DHEW), Washington, D.C.
BUREAU NO 020193
PUB DATE Jun 75
GRANT OEG-0-72-0895
NOTE 68p.; For related document, see ED 078 681

EDRS PRICE MF-$0.83 HC-$3.50 Plus Postage.
DESCRIPTORS College Students; *Computer Assisted Instruction; Computers; *Concept Teaching; Engineering Education; High School Students; Individual Instruction; *Mathematical Concepts; Models; *Problem Solving; Program Descriptions; Programed Tutoring; Programing; Systems Concepts; *Tutorial Programs
IDENTIFIERS CAILD; CAI System for Logic Circuit Debugging and Testing; *Generative CAI; Generative Tutor; Interactive Student LISP Environment; ISLE; LISP; Machine Language Tutor; MALT

ABSTRACT

This final report summarizes research findings and presents a model for generative computer assisted instruction (CAI) with respect to its usefulness in the classroom environment. Methods used to individualize instruction, and the evolution of a procedure used to select a concept for presentation to a student with the generative CAI system are described. The model served as the basis for the design of a CAI system, teaching problem-solving in an introductory course in digital systems design. The system individualizes the instruction which each student receives in accordance with its record of his past performance. In addition, a heuristic technique is used to determine the best path for each student through the tree of course concepts. The refinement of this method of concept selection is described. An evaluation of the generative CAI system and results of classroom usage are also presented. (Author/DAG)

Final Report

Project No. 020193
Grant No. OEG-0-72-0895

An Intelligent CAI Monitor and Generative Tutor

Elliot B. Koffman
James Perry

University of Connecticut
Department of Electrical Engineering
and Computer Science
Storrs, Connecticut 06268

June 1975

The research reported herin was performed pursuant to a grant with the National Institute of Education, U.S. Department of Health, Education and Welfare. Contractors undertaking such projects under Government sponsorship are encouraged to express freely their professional judgement in the conduct of the project. Points of view or opinions stated do not, therefore, necessarily represent official National Institute of Education position or policy.

U. S. Department of
Health, Education and Welfare

National Institute of Education

ABSTRACT

This paper describes a model for generative computer-assisted instruction. This model has served as the basis for the design of a CAI system used to teach problem-solving in an introductory course in digital systems design. The system individualizes the instruction which each student receives in accordance with its record of his past performance. In addition, a heuristic technique is used to determine the best path for each student through the tree of course concepts. The refinement of this method of concept selection is described. An evaluation of the GCAI system and results of classroom usage are also presented.

TABLE OF CONTENTS

TABLE OF CONTENTS
(continued)

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Over the past few years, we have been exploring the concept of generative computer assisted instruction (GCAI). This paper summarizes the findings of this research with respect to the usefulness of a generative CAI system in a classroom environment. The paper describes methods used to individualize instruction and the evolution of a procedure used to select a concept for presentation to a student working with the CAI system.

This paper is concerned with problem oriented systems. In a typical drill and practice system values are generated and substituted for variables in a prestored question format. The values of the variables are constrained so that the resulting problem is meaningful and of an appropriate difficulty level. In this way some of the semantic content of a question is provided. In a problem oriented generative system, a problem is generated from a grammar or other suitable structure. Both the syntax and semantics of a problem are determined by the system. Thus, a richer variety of problems can be generated. In addition, the system has the potential of controlling the generation process so that it can tailor the difficulty and area of emphasis to suit the individual student.

A generative CAI system must have the capability of solving the problems that it generates. Usually, the problems will cover a specific approach to design or a solution algorithm. They will be more complex and involved than the manipulations performed in a drill and practice environment. Indeed, the incorporation of problem solving capability and semantic knowledge indicate the need of an Artificial Intelligence (AI) approach to generative CAI design. However, the problem solvers needed in CAI applications are less difficult to design for the following reasons: 1) they are concerned with specific problem types in a specific subject area 2) they are generally more algorithmic than heuristic 3) they are supplied with problem generation information including parameters necessary for solution and do not have to extract this information from the problem representation. Bobrow's STUDENT (1) and Winograd's SHRDLU (2), for example, must first interpret the problem statement and extract essential information prior to obtaining the solution.

Nevertheless, generative CAI systems have benefited and can benefit further from research in artificial intelligence on natural language understanding, question - answering, problem solving, and automatic programming (3). Specifically, AI can be beneficial in the design of more general solution routines so

that it will not be necessary to implement a solution routine for relatively small problem classes. AI techniques can also contribute to the construction of solution routines to help achieve more powerful generation systems. On the other hand, study of generative CAI and generative techniques can make a contribution to artificial intelligence by providing results, for example, on ways of identifying and using control (generation) information to design efficient and practical AI systems.

2. MODEL FOR GENERATIVE CAI (GCAI)

A. Introduction

Figure 1 is a block diagram of a GCAI system. To individualize instruction, this system needs a model of the course and the student - the concept tree and student record respectively. After a concept has been selected and the appropriate level of difficulty determined, a problem is generated and presented to the student. The subsequent interaction and monitoring of the student's solution is a function of his performance. Each of the components in Figure 1 will be discussed below.

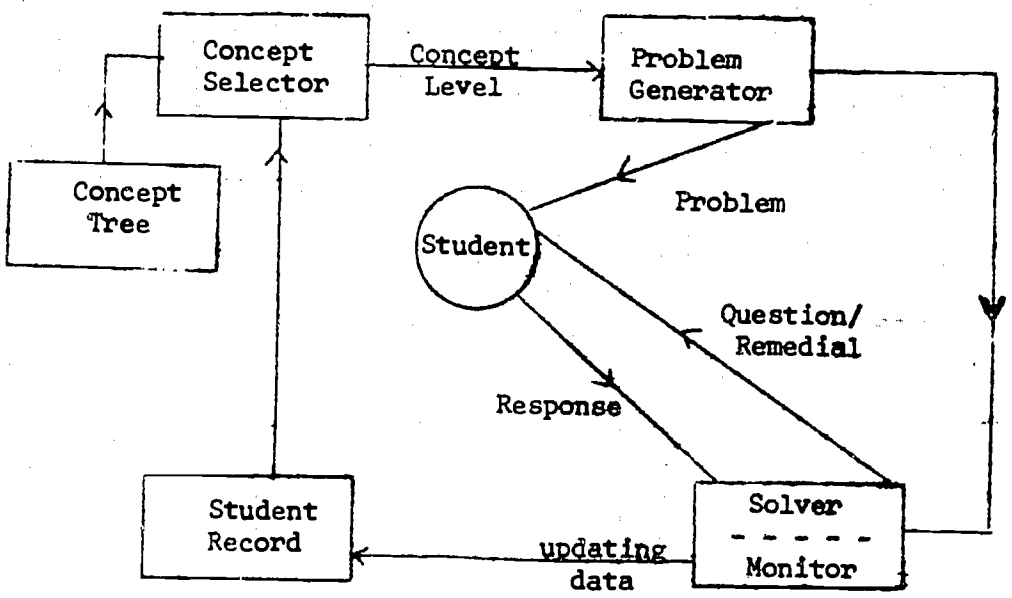


Figure 1. System Block Diagram

B. Course Structure

A course can be modeled as a hierarchical tree structure in which each node represents a concept and corresponds to the following information:

1) concept number
2) concept name
3) a list of doubles; the first element of each pair is the number of a prerequisite concept and the second, a flag indicating whether or not a prerequisite may be called as a subroutine
4) the plateau for that concept (how far up on the tree it is), which implies its relative complexity
5) the name of the problem generator for that concept
6) the name of the solution routine
7) a list of parameters which are passed from the problem generator to the solution routine

For example, (C7, BINARY MULTIPLICATION, ((C4 0), (C5 1)), 2, PROB1, BINMUL, (levC7, levC5, multiplicand, multiplier)) means that C7 deals with binary multiplication. There are two prerequisites. One of the prerequisites may be called as a subroutine, (C5 - binary addition). C7 is on plateau 2 of the concept tree. The parameters passed from the problem generator PROB1 to the solution routine, BINMUL, are the student's levels of proficiency in concepts C7 and C5, together with the multiplicand and multiplier for the generated problem.

The level of proficiency in C7 ($.5 < levC7 < 3$) determines the difficulty of the problem to be generated and the degree of instruction and monitoring to be received by the student while working on a binary multiplication problem. The parameter levC5 is needed in the event concept C5 is called as a subroutine.

This representation is general and has been used in a system for digital systems design (4) and a system for high school algebra (5). In a varied form it has also been used for a machine language programming GCAI system (6).

The digital systems design GCAI course contains twenty-one concepts beginning with basic logical operations and number conversions (Plateaus 1, 2), combinational design and timing diagrams (plateau 3), minimization techniques (plateau 4), sequential design (plateaus 5,6) and register transfer operations (plateau 7).

The arcs of the tree structure (concept tree) represent prerequisite or subconcept relations; the latter indicated by a flag. More generally, the arcs

can represent other relations of interest. In machine language programming (6), the concepts are small sections of a program called input primitives, processing primitives, and output primitives. The relation of interest is concatenation. The nodes of the concept tree represent the primitives and the arcs represent possible concatenations of the primitives to form more complex problems.

C. Problem Generator

Only two problem generator routines are used for the first thirteen concepts in the digital systems course. This is due to the fact that there is a great deal of similarity between the parameters needed for these routines. Several of these routines require a variable number of binary, octal, decimal, or hexadecimal character strings as parameters.

A viable model for a problem generator is that of a probabilistic grammar. A context free probabilistic grammar is a formal grammar whose rewrite rules have a single nonterminal symbol on the left (generating symbol). Moreover, each rewrite rule has a non-zero probability associated with it such that the sum of the probabilities of the rules with the same generating symbol is one. In order to obtain problems of varying degrees of complexity, the probability of each rewrite rule is made a function of a student's level of proficiency in the concept being studied; rewrite rules leading to more difficult problems become more likely as a student's proficiency increases. The probabilities can also be made functions of the estimated difficulty of the partially generated problem. For example, the depth of nesting in a logical expression is one indicator of difficulty. The probability of rewrite rules which tend to produce nested subexpressions exceeding a preset threshold should drop to zero. This makes the probabilities context sensitive and keeps the problems from getting out of hand.

The grammar for the problem generator associated with each concept consists of a 7-tuple of the following format:

$$G = (S,N,T,R,P,Z,D)$$

where S is the starting symbol, N is the set of nonterminal symbols, T the set of terminal symbols, R the set of rewrite rules. P is an array of probabilities with three columns since it was preferred to distinguish between only three categories of problem difficulty. The student's previous level of performance in a problem determines which column of probabilities will be used during the

generation of a problem. Each row in P corresponds to a rewrite rule. The probabilities associated with rewrite rules having the same generating symbol sum to one over each column of the array.

D is a function which may be used to calculate the difficulty of the partial output string and Z is a vector of difficulty thresholds as mentioned above. For most problem classes, neither D nor Z were deemed necessary.

Table 1 is an example of a grammar which generates logical expressions for the concepts which teach truth table formation and the analysis of sequential circuits.

The first column in the array P is the vector of probabilities for beginning students, the second column is for intermediate level students, and the third column is for advanced students. The harder operators ↑ (nand), ↓ (nor), and ⊕ (exclusive-or) become more likely as a student's proficiency increases. Similarly, the number of distinct variables likely to appear in an expression increases with student proficiency. Rules R_1/R_2 which expand the expression also become more likely. The vector Z specifies that the probability of these two rules drops to zero when the length of the expression exceeds the student's proficiency level by a factor of twenty.

Table 1: Probabilistic Grammar Example

N = {A,*}

T = {p,q,r,s ,^,V,↑,↓,⊕}

S → A

	P			Z
R_1: A → (A*A)	.25	.3	.35	20
R_2: A → (A)	.25	.3	.35	20
R_3: A → p	.25	.14	.07	1000
R_4: A → q	.25	.13	.07	1000
R_5: A → r	0	.13	.08	1000
R_6: A → s	0	0	.08	1000
R_7: * → ^	.5	.2	0	1000
R_8: * →	.5	.2	0	1000
R_9: * → ↑	0	.2	.34	1000
R_{10}:* → ↓	0	.2	.33	1000
R_{11}:* → ⊕	0	.2	.33	1000

$D = \frac{\text{length (expression)}}{\text{proficiency level}}$

D. Problem Solvers and Solution Monitors

In order to solve the problems generated, a set of augmented solution algorithms is provided, one per concept. These algorithms were augmented with additional information. Each distinct subtask in the algorithms is identified and provided with: (1) generation threshold, (2) question format, (3) remedial format set, (4) answer comparator, (5) increment and decrement multipliers.

The generation threshold indicates a proficiency level beyond which a student's solution to the subtask will not be monitored.

The question format is a skeletal pattern into which problem dependent variables will be inserted. The formats are used for generating questions concerning each subtask.

The remedial set is utilized when a student's answer does not match the system's solution. It usually consists of formats which provide some explanation of the correct solution procedure in the context of the current problem. Each format has an associated threshold level. If a student has a proficiency level greater than the threshold, the remedial statement is not printed out. Hence, the completeness of the explanation provided by the remedial set decreases as student proficiency increases.

The answer comparator determines the degree of correctness of the student solution. In most cases, the degree of correctness can be determined by matching the solution string derived by the system with that provided by the student. For some algorithms the comparator consists of an analysis routine which determines the type of error being made by the student in order to provide more meaningful remedial commentary. The analysis routine may make use of portions of the solution algorithm to assist in characterizing the student's error. This allows the system's procedural knowledge to be used to make remedial commentary more meaningful.

The increment and decrement multipliers determine the amount of increase in proficiency level for each correct solution to a subtask and the decrease for each incorrect solution. In this way, each subtask is weighted to produce the desired degree of change in level. As part of the increment and decrement, a factor based on the student's prior performance in a concept is calculated just before the algorithm is called. This factor tends to produce increments which are larger than decrements.

for students who are performing well and vice-versa for students performing poorly. Thus, good students are penalized less for an occasional error and poorer students are required to become more consistently correct in order to advance to the next higher level range.

The augmented solution algorithm leads a beginning student carefully through a solution procedure while providing more freedom and less interaction for advanced students. The degree of freedom and monitoring is dynamically adjusted as a student's level of performance changes. Table 2 gives examples of the varying degree of interaction possible. Reference 4 describes problem generation and solution in greater detail.

TABLE 2: Student Interaction in Problem Solution

Initial Level Range 0-1

Form the logical AND of 7345,2206 in the base 8:
Taking corresponding pairs of digits starting at the right:
What is the binary equivalent of 5?
101
What is the binary equivalent of 6?
111
No. The binary equivalent of 6 is 110.
What is the logical AND of 101,110?
100
What is the logical AND of 5, 6?
4 (Note: Here student's level becomes >1)
What is the logical AND of 4,0?...

Initial Level Range 1-2

Form the logical AND of A2.4,1.C4 in the base 16:
Modify A2.4 by adding trailing and/or leading zeros.
0A2.4
No. A2.4 should be changed to A2.40
Modify 1.C4 by adding trailing and/or leading zeros.
01.C4
Form the logical AND of A2.40,01.C4 in base 16:
Taking corresponding pairs of digits starting at the right:
What is the logical AND of 0,4?
0
What is the logical AND of 4,C?...

Initial Level Range 2-3

Form the logical AND of C3.4,1.493 in the base 16:
What is the logical AND of C3.400,01.493?
03.400
No. The logical AND of C3.400, 01.493 is 01.400

E. Review Mode and Example Mode

Two additional and extremely attractive benefits of this generative approach to CAI are the capability to operate in a review mode and an example mode. After the concept has been mastered (proficiency level >3), all subtasks will be solved by the system and no solution monitoring will take place. Only summarizing data and sub-concept solutions will be printed out. A student can use this mode to quickly obtain solutions to several of the more difficult problems for later self-study.

When dealing with a new concept, the student may be uncertain as to some details of the solution procedure. There is occasionally some ambiguity in questions which are posed. Also, while variable formats are allowed for the student's response, the system is usually better able to handle certain formats than others. To alleviate these difficulties, an example mode may be entered by setting a flag which causes each question posed to the student to be followed by a "write" statement rather than a "read" statement. The net effect is for the system to correctly answer all of its own questions instead of monitoring the student's response. The student is able to familiarize himself with the content of the concept and see what will be expected of him. With each question, the proficiency level temporarily increases so that the student observes a change in solution monitoring similar to that which would be obtained during normal interaction with the system. The example mode option is made available prior to a student's initial use of each concept and is almost always used.

3. SOME CHOICE FUNCTIONS FOR CAI

A. Introduction

The systems concept selector selects a concept for study based on a student's past performance. The concept selector is the interface between the student and the tree of course concepts. It determines which concept is 'best' for a student at any given time, and it traces an 'optimal' path for each student through the course concept tree.

It is difficult to measure the 'goodness' of a particular concept choice and there would probably be no consenus among educators as to which particular concept is best for a student at a particular point in time. The goals of concept selection, however, are necessarily clear. The time spent by each student in mastering the

CAI course should be kept to a minimum. Thus, he should not be required to waste time on a concept already mastered; nor should he be expected to solve a problem if he has not shown mastery of its prerequisite concepts. The intent is to pace him through the course concepts as quickly as possible without confusing him.

There have been many studies on the effect of learner versus system control in a CAI environment. These studies seem somewhat inconclusive to date. Rather than restrict the student to one mode of control, the approach taken was to attempt to do the best job of concept selection possible, but to always provide the student the opportunity to veto any individual system selection or to take control and bypass system selection entirely.

The task of concept selection is done in two phases: the first phase determines the current plateau of the concept tree; the second selects the actual concept to be worked.

B. Determining the Current Plateau

A record of past performance is maintained for each student. This student model is shown in Table 3. The twelve items in the table are saved for each of the twenty-one concepts of the digital systems design course. As mentioned above, the difficulty of the problem generated, and the amount of instruction and monitoring a student receives is proportional to his level of competence in that concept. A student's level of competence is a real number between .5 and 3. It is incremented by a small amount ($|\Delta L| < .28$) for each correct answer and decremented for each incorrect answer. As a student will normally answer more questions correctly than incorrectly, his level should gradually increase.

In addition, for each student, the system saves the current plateau and a performance regulator, called the master average. The master average varies between 1.4 and 2.5. A significant increase or decrease in concept level ($|\Delta L| > .5$) results in a proportional change in the master average in the opposite direction. Consistently good performance will result in a low master average. The master average is updated after each problem is completed.

TABLE 3: Student Record

1 The present level for each concept (L)
2 A weighted average of level changes for each concept (WTAVG)
3 Last level change (ΔL)
4 The date each concept was last worked
5 A sequence number indicating the ordering of system selection
6 # of times concepts worked in 0-1 range
7 # of times concepts worked in 1-2 range.
8 # of times concepts worked in 2-3 range
9 Date number of last time each concept worked
10 # of times student rejected each concept
11 # of times each concept was selected by system and accepted by student

When a student's average level in all the concepts at his current plateau exceeds his master average, he is eligible to proceed to the next plateau. Consequently, the better a student performs, the faster he will progress up through the concept tree. It is possible that a concept will only be worked once or even not at all. For this reason, a post-test is given which consists of a challenging question for each concept. As a result of the post-test, the system advances him to the next plateau or may require him to review each concept whose corresponding post-test question was answered incorrectly. The decision whether or not to review a concept is based on the number of times he has worked that concept, his level in that concept, and his master average.

If a student is advanced to the next plateau, a pre-test is given to initialize the levels for the concepts on that plateau. The pre-test consists of a set of three or four questions for each concept. The questions get progressively more difficult. A student is given two tries at each question and branched to the next set of questions if both answers are incorrect. The starting level for each concept is a function of how many of the set of questions dealing with that concept were answered correctly. The starting level ranges from .5 to 1.0.

The pre-test for the first plateau is also used to initialize the master average. The more questions answered correctly in the first pre-test, the lower will be the starting master average.

C. Concept Selection

There are three or four concepts on most plateaus of the concept tree. A preliminary check is first made to ensure that the average level of the prerequisites for each concept exceeds the student's master average. If this is not the case,

this concept is replaced as a candidate by its prerequisite with the lowest level. This provides an additional opportunity for review in case the student has been progressing too rapidly.

Statistics are available from each student's record (refer to Table 3) concerning the last date each concept was used, its current level, the last change in level, and the number of times the concept has been selected by the system or rejected by the student in the past. From this data, many parameter values can be calculated for each concept and input to the concept selector. The concept selector attempts to predict which concept will advance the student the most in the course.

A scoring polynomial as used by Samuel (7) in his well-known Checkers program was the basis for the concept selector. A scoring polynomial consists of a set of parameter values to be calculated for each considered move and associated coefficients or weights. A preliminary training period was undergone in order to determine a good set of coefficient values. During actual play, the move which scored the highest was always made. In a similar manner, the concept which scores the highest among those being considered should be selected for presentation to the student.

Samuel's model assumes that there is a single "best move" at any point in the game. However, the "best concept" for a given student may depend on his particular learner preferences and personality traits. Consequently, it was decided to provide for four separate sets of polynomials and use a student's past performance to determine which set appeared best suited to him.

Two training periods were undergone by the system in order to define four scoring polynomials. Initially, a single scoring polynomial which appeared to work reasonably well was derived. This polynomial is shown in Table 4.

TABLE 4: Original Independent Variables

Var. 1	$(2*\Delta L_I)^2*w1$	w1=1 if $L_I<0$; =0 if $L_I\geq 0$
Var. 2	$(Q-Q_I)/(Q-Q_M)*w2$	w2=1.2 if $Q_I>0$; =3 if $Q_I=0$ (Concept never worked)
Var. 3	R*w3	R=# of uses of a concept at present level range (0-1, 1-2, 2-3)/total number of times worked; w3=-1
Var. 4	$\lvert\Delta L_I\rvert/L_I*w4$	w4=1

Table 4: Continued

Var. 5	$(T_D+T_P)*w5$	w5=.1
Var. 6	$(3-L_I)*w6$	w6=1 if $L_I<3$
		=-25 if $L_I\geq 3$

Level of concept I;

L_I: Concept level; ΔL; Last level change; for concept I
I:: Current concept being considered; Q_M: Minimum sequence number of all concepts on present plateau
Q:: Current sequence number; Q_I: Sequence number of last time concept I was worked
T_D: Number of prerequisites directly callable as subproblems; T_P: Total number of prerequisites

The rationale for the parameters of the polynomial was as follows. Parameter 1 contributes to the weighted sum if the student's level in a concept has just dropped. Obviously, he has not mastered this concept. A decrease in level normally results in more system monitoring and help. Parameter 2 favors the concept that has been waiting the longest. Parameter 3 favors a concept that is in a new or relatively unstable level range ($R << 1$). Parameter 4 favors a concept whose level is changing rapidly. Parameter 5 favors the concept which is based on the most prerequisite concepts since working this concept will help to review earlier concepts. Parameter 6 favors the concept which currently has the lowest level. When a concept has been mastered (level > 3) a large decrement is added to that concept's score.

Observation of students in the course showed that they initially relied heavily on the concept selector. As the course progressed, they assumed more of the task of concept selection. Part of the reason was their increased familiarity with the course contents and knowledge of their weaker areas. This aspect of the situation was encouraged as it normally resulted in less waiting time between concepts and the students seemed quite proficient at determining their own needs.

For two semesters, data was collected on all student and system concept selections (8). A record was kept of parameter values for each considered concept, the actual concept selected, student acceptance or rejection, and subsequent change in level for that concept. Fourteen additional parameters were added to the original six (see Table 5) and a multiple linear regression analysis was performed to calculate optimal coefficients for the twenty independent variables. The dependent variable value associated with each concept selection was made proportional to the resulting change in level as the goal of concept selection was to advance the student as rapidly as possible.

TABLE 5: New Independent Variables

Var. 1-Var 6 same as Table 4
Var. 7 ΔL_I (last level change)
Var. 8 $WTAVG_I$ (weighted average level change)
Var. 9 L_I (level)
Var. 10 # of days since last worked concept I
Var. 11 # of days since last used system
Var. 12 Total of # of concept worked
Var. 13 Total # of days worked
Var. 14 Var. 12/Var. 13
Var. 15 # of times student worked concept I
Var. 16 # of times student rejected concept I
Var. 17 # of times system selected concept I
Var. 18 # of times student selected concept I
Var. 19 Var. 16/Var. 17
Var. 20 # of times worked concept I at present level range

This intermediate step produced a single scoring polynomial which should theoretically be better than the original polynomial with only six variables and non-optimal coefficients. To compare these polynomials, a second regression analysis was performed using only the original six independent variables as predictors. This analysis produced a multiple correlation coefficient of .18 as compared to .48 for the expanded scoring polynomial.

The final step was to perform a cluster analysis on the data collected in an attempt to identify four groups of similar students. The results are shown in Table 6 along with the multiple correlation coefficient associated with each group. For all except group #3, this coefficient is higher than that associated with the population as a whole (.61, .61, .43, .53 versus .48) and should thus lead to improved concept selection for the majority of students.

TABLE 6: Regressions on Groups

Group #1			Group #2			Group #3			Group #4		
Var.#	$\hat{\beta}_I$	F_I	Var.#	$\hat{\beta}_I$	F_I	Var.#	$\hat{\beta}_I$	F_I	Var.#	$\hat{\beta}_I$	F_I
19	-1.226	100.884	19	-.895	30.696	7	.561	30.642	19	-1.348	69.251
5	.427	35.684	2	.262	12.474	19	-.814	24.520	7	.375	10.848
9	-.467	14.437	20	.526	11.207	5	.301	23.871	5	.174	7.948
2	.149	12.690	3	1.870	10.959	9	-.531	16.423	16	.125	7.795
16	.063	11.638	15	-.410	8.716	10	.007	8.416	1	.136	6.042
4	.551	11.230	5	.287	8.524	18	.126	7.887	9	-.214	5.021
3	.482	7.067	9	-.690	8.106	12	-.003	5.091	3	.315	4.471
12	-.002	3.674	18	.240	6.572	3	.346	4.547	8	.350	3.037
7	.227	3.199	6	-.497	4.812	16	.072	3.354	6	.067	2.207
6	-.148	1.845	4	.406	2.923	6	-.167	3.134	15	-.012	1.067
	p = .61			p = .61			p = .43			p = .48	

Some observations from this table are that variable 19 appears at or near the top of all groups. Variables 5 and 9 appear near the top of all groups. Variable 2 has a relatively high significance in groups 1 and 2 while variable 7 has a relatively high significance in groups 3 and 4.

The complete concept selection cycle consists of the following steps:

1. Determine the student's current group
2. Find the current plateau
3. Identify candidate concepts
4. Evaluate each concept using scoring polynomial for student's group
5. Present highest scoring concept
6. Update scoring polynomial compatability ratios

Steps 2) through 5) have been discussed previously. Steps 1) and 6) are accomplished in the following manner. A compatability ratio is maintained by student for each scoring polynomial. The student is placed in the group which has the highest compatibility ratio. The concept which would be chosen by each group is determined, and the concept selected by the student's current group is presented. After completion, the compatibility ratios of all groups which selected this concept are updated to reflect the student's change in performance level. In the event the student rejected the concept selected and chose another considered concept, the compatability ratios of all groups which chose this concept would be updated instead.

The next section will discuss results of classroom experience with this generative CAI system.

4. RESULTS AND CONCLUSIONS

It is a rather involved problem to control an educational experiment carefully enough to obtain a meaningful comparison of one teaching method versus another. In selecting control groups, there are many variables which enter in such as motivation and aptitude for the course, inherent favorable or unfavorable biases towards computers, and personality traits which are difficult to measure. Most of our enthusiasm and interest in this project has been spurred by conversations with students who have used this system and an examination of student questionnaires which rate the system very highly as an educational tool (more on this later).

In any event, an experiment was set up during the fall semester of 1973 in which two randomly assigned classes (determined by the unversity class scheduling algorithm) of approximately thirty students each were designated as GCAI and control group respectively. Both classes had the same instructor and used the same textbook. The basic differences between the course structure for each group are shown below:

GCAI Class	Control Class
2 hours of lecture/week	3 hours of lecture/week
homework problems done thru GCAI	conventional homework assignments
seven 20-minute quizzes	two 75-minute exams

The GCAI students were required to take short quizzes after each plateau. These were administered on an individual basis and were primarily used to encourage students to keep a reasonable pace.

The GCAI system was implemented on an IBM 360/65 computer. Students worked at their own pace and were free to use it whenever the time-sharing system, CPS (9), was operating.

Prior to the start of the course, both groups were given the same pre-test which was a simplified version of a previous semester's final exam. The mean score for the control group was 25 out of a possible score of 100; the GCAI class mean score was only 9. Median scores for each group were identical to the mean scores.

Earlier analysis of past student performance in this course had shown the single best predictor of success to be the student's overall quality point ratio (QPR) or cumulative average of grades since entering the university. The mean QPR for the control group was 2.8 out of a possible 4.0 versus only 2.5 for the GCAI group.

Both groups of students were given the same final exam; however, there was no significant difference in their performance (mean score GCAI - 115/150, Control 113/150). Since the GCAI groupt spent 1/3 less time in class and was not as well prepared for the course, this is a very favorable result.

This course is a prerequisite for an advanced digital design course. There were approximately ten students from each group in this course during the following semester. Again there was no significant difference between the grades received by the GCAI students and the control group.

Three different instructors have used the GCAI system over a period of two years. They generally feel that GCAI is beneficial in that it frees them from having to discuss at length routine problems and techniques. Instead they are able to concentrate on more difficult concepts and intuitive approaches to design as students are receiving practice in the basic techniques and algorithms through GCAI.

As has been mentioned, student reaction to GCAI is generally quite favorable. The results of a questionnaire returned by sixty-four students are summarized in Table 7.

TABLE 7: Student Questionnaire

1. I preferred this system to conventional homework assignments

SD	D	U	A	SA
3%	14%	8%	43%	32%

2. CAI is an inefficient use of the student's time

20%	41%	17%	16%	6%

3. I was concerned that I might not be understanding the material

11%	44%	17%	24%	3%

4. CAI made it possible for me to learn quickly

2%	16%	17%	60%	5%

5. The CAI system did a good job of selecting concepts

9%	29%	23%	36%	3%

6. I found the "example mode" feature useful

0%	3%	9%	50%	38%

SD: Strongly disagree; D: Disagree; U: Uncertain; A: Agree; SA: Strongly agree

The only question which is not responded to in a manner which indicates strong acceptance is #5 dealing with the goodness of concept selection. Here 39% of the students responded favorably and 38% unfavorably. We feel this is a reasonable achievement in a very difficult area of decision making where there is clearly no single right or wrong solution. This is a somewhat better rating than that received by the earlier scoring polynomial.

In conclusion, we feel as do the majority of students who have used GCAI; it provides an opportunity to learn by doing and is an effective way to teach problem-oriented material. Students receive immediate feedback and learn from their mistakes. GCAI helps a student master the basic concepts and enables the student and instructor to concentrate on more advanced material.

II. GENERATIVE CAI IN MACHINE LANGUAGE PROGRAMMING

1. Introduction

Prior reports have described the MALT System (MAchine Language Tutor - See References 6, 10.). MALT is concerned with teaching machine-language programming for a simplified version of the Digital Equipment Corporation PDP-8 minicomputer. The instruction set of the hypothetical machine is virtually identical to that of the PDP-8. The major difference is that this computer has only 400_8 memory registers; consequently, each register is directly accessible. This means that students can learn the fundamentals of machine-language programming without the added complexity of memory page consideration.

MALT is a generative CAI system in two important senses. First, it creates its own sample programming problems using a variety of heuristic techniques. It is not dependent upon the course author for a complete supply of ready-made problems and their solutions. Instead, by beginning with only a series of basic problem elements or sentences, it generates a problem that is consistent with the user's present ability.

Another important way in which the system is truly generative is its ability to design a solution program for the problem that it has generated. By using basic algorithms supplied by the course instructor, the system can produce the actual machine code of a solution program. This implies that the system is quite flexible, since later alteration and extensions involve only the addition of new programming algorithms, not massive system reorganization.

MALT attempts, through constant monitoring of the student's program to determine not only the existence of logical errors, but also their location in the program. This ability enables the system to be much like the human teacher; that is, it can note and correct logical errors before they develop into undesirable programming habits.

The system attempts to tailor its presentation to fit the abilities of the students. Any problem that is generated is designed to provide the student with a challenge, while not being beyond his capabilities. The dialogue initiated by the system will also be governed by the user's performance. A beginning student will receive a wide variety of hints and suggestions for the design of his program. Also, his errors will result in quite explicit and complete remedial messages. As the student progresses through the material, he will receive less system information and be given more freedom in his programming actions. In

addition, the more elementary sub-tasks will be programmed for him. When the student achieves high proficiency, the system can function purely as a problem solver in that all programs will be generated by it. This facility is useful if a student desires to study examples of advanced problems and their corresponding solution programs.

As the system questions the student, it is constantly developing its own solution program for comparison with the student's program. In this way, a given programming concept is rarely presented the same way more than once to a particular student. The student's enjoyment of the system is thereby greatly enhanced because he receives new dialogue with every problem.

The actual operation of MALT is straightforward. After the student identifies himself to the system, his records are obtained and evaluated. These records determine the difficulty and content of the problem generated and the amount of instructional guidance that the student will receive. Next a sample programming problem suited to his abilities is generated. To help him design his program, the system will then develop a logic chart or list of sub-tasks. These sub-tasks break the problem into a series of smaller, more manageable steps and are of great help to the novice programmer.

As the student undertakes each sub-task in the programming process, a corresponding concept routine is entered by the CAI system, which guides the student through the construction of that part of his program. During this phase, the student is constantly being given feedback as to the correctness of his program. If his program introduces logical errors, the system will point these out and offer helpful suggestions for their correction. If the system checking procedures determine that the student might benefit from observing his program in operation, it also has the capability to simulate statement by statement program execution

Figure 2 is a block diagram of the Malt system. Table 8 provides a complete example of the operation of MALT. The comments in square brackets have been added to clarify this example for anyone not familiar with the PDP-8 instruction set. Each student response is preceded by a dash. The dialogue shown is that which would be received by a beginning student.

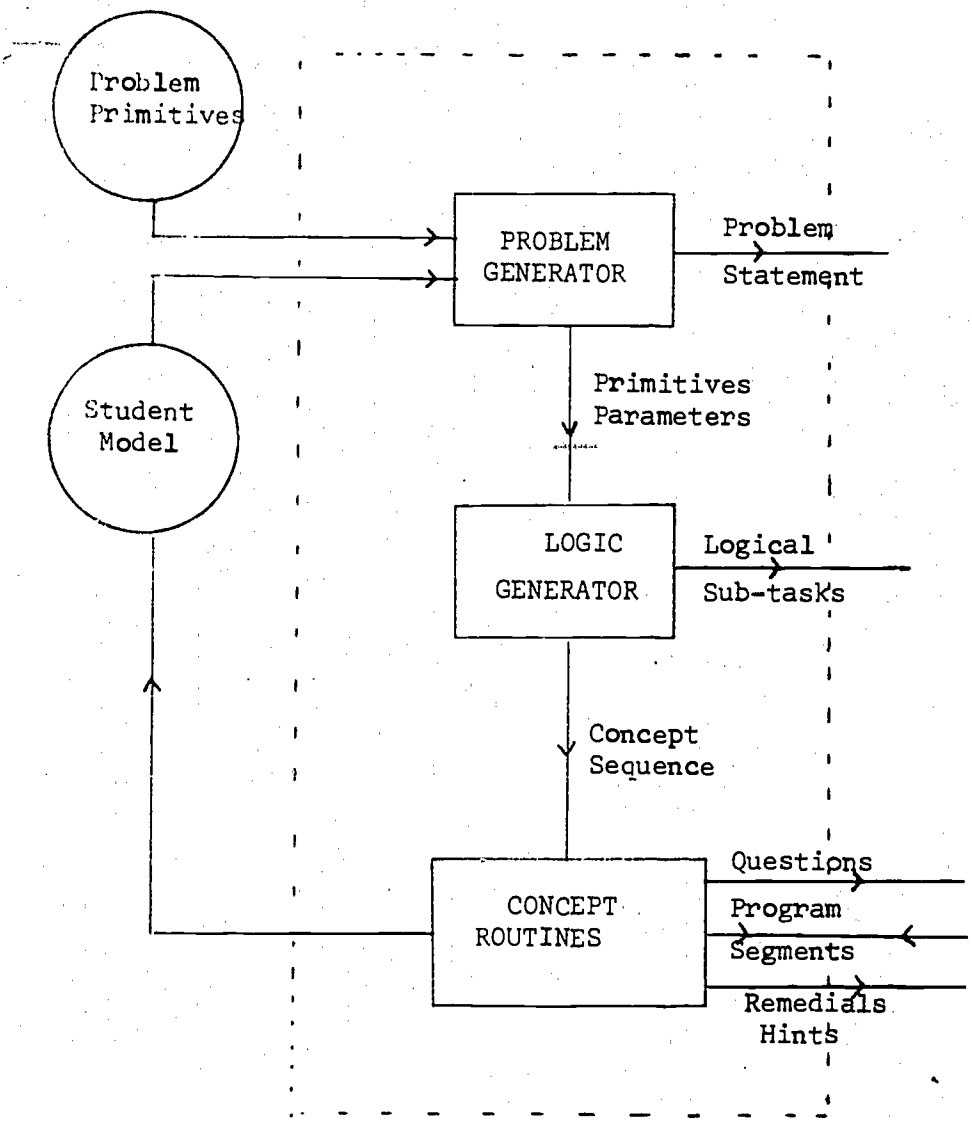


Figure 2 - MALT Block Diagram

TABLE 8: MALT Sample Dialogue

Your problem is to write a program which will:
Read in 20 (octal) ASCII characters and store them in registers 232 through 252.
Form the sum of registers 232 thru 252 in the accumulator.
If this results in a nonzero LINK, stop with the (ACC) = 7777, otherwise stop with (ACC) = 0000.

Is this problem OK?
—yes

Here are the sub-tasks for the first line.
1) Initialize a ptr to register 232.
2) Initialize a ctr with the value of −20(octal).
3) Read in a character.
4) Store it away using the ptr.
5) Update the ptr.
6) Update the ctr and if it is not zero, jump back to start of loop.

SUB-TASK 1:
Since we may need a ptr later, we will store the ptr value in memory and move it to a register which we will use as the actual ptr.
Use M377 to store the ptr value, and M376 as the ptr.
M377:
—0232
Now, get the ptr value:
M000: [location of 1st instruction]
—1376 [add in contents of M376]
WRONG!
"TAD 377" is coded as :1377
And deposit it into the ptr:
M001:
—3376 [Deposit ACC into M376, clear ACC.]

SUB-TASK 2:
We will store the ctr value in memory and then move it into another register for use as the ctr.
Use register 375 to store the ctr value and register 374 as the ctr.
M375:
—7757
7757 can't be used as the ctr value. Try again!
M375:
—7760 [−20 in 2's complement form]
First, get the ctr value:
M002:
—1375
And deposit it into the ctr.
M003:
—3374

SUB-TASK 3:
First, check the reader flag.
M004:
—5031
NO!
To check the reader flag is 6031.
Next, jump back one register if it's not ready.
M005:
—5004 [JUMP to M004.]
Now, operate the reader.
M006:
—6036

SUB-TASK 4:
M007:
—3376
Wrong!
"DCA I PTR" is :3776 [This should be an indirect deposit.]

SUB-TASK 5:
OK, update the ptr.
M010:
—2376 [Increment M376 and skip next instruction if M376 contains 0000.]

SUB-TASK 6:
Now, increment the ctr.
M011:
—2374
And if not zero, jump back to start of loop.
M012:
—5002
Wrong!
The loop starts in register 004 so it is 5004

2. Results and Conclusions

The system has been implemented in the CPS (9) language on the IBM 360/365 at the University of Connecticut Computer Center. Students can use this system whenever they desire. There is also a batch mode simulator of this computer which they use for class projects of a more ambitious nature.

Student reaction to MALT has been very favorable. They feel this system helps to bridge the gap between what they have learned in class, or from the textbook, and what they need to know to program independently in batch-mode. Normally, students spent two weeks using MALT and were then given a week to get a rather sizeable problem coded and running in batch-mode. This proved to be significantly easier for students who had used MALT than for those who had not (See question 2 in Table 9).

A questionnaire was distributed to the classes using MALT. The results of this questionnaire are tabulated in Table 9. It appears that the students feel that this experience was beneficial and good preparation for learning to program independently. On the whole, students were not bothered by the fact that MALT requires them to adhere to a particular "flowchart". As indicated by question seven, improvements could be made to the algorithm which determines that a generated problem is sufficiently different from previous problems presented to that student. Ninety students responded to the questionnaire.

TABLE 9: MALT Student Evaluation

For question 1-9 the percentage of students giving the following responses are tabulated

	Strongly Disagree	Disagree	Uncertain	Agree	Strongly Agree
1. The System was useful in introducing me to machine language programming.	2	2	4	56	35
2. It was relatively easy to learn to use the batch version of the assembler since I had been introduced to programming concepts through MALT	0	6	18	50	26
3. Since the sub-tasks were always laid out for me, I felt very constrainted using MALT	5	49	21	25	0
4. Because the sub-tasks were laid out, I only learned the mechanics of programming and really didn't understand what was going on.	9	46	31	11	3
5. The approach taken in printing out the sub-tasks was good as it taught me how to organize a machine-language program.	0	4	20	62	14
6. The problem became more difficult as my level increased.	2	11	21	60	6
7. There was a good variety in the problems I received in MALT.	6	28	17	48	1
8. In general, I enjoyed the interaction with MALT.	1	0	14	69	16

Overall, we feel that MALT is an effective demonstration of what can be accomplished in CAI with the limited use of AI techniques. It should be stressed that MALT's design has been influenced by AI research, but certainly much more could be done in the way of incorporating AI Research in problem solving and program synthesis. The desire to produce a working system with reasonable response time on an existing time-sharing system precluded this possibility. Hopefully, MALT will challenge others with an interest in CAI and AI to purse this goal further.

III. COMPUTER ASSISTANCE IN THE DIGITAL LABORATORY

1. Introduction

Most of the CAI systems implemented so far teach subject areas that are normally being taught in a classroom environment. Very little work has been done in the area of laboratory instruction. Neal and Meller [11] have implemented a system which teaches students to operate laboratory electronic instruments.

In the field of computer science, the student's backgrounds are quite diverse. Many of them come from fields other than electrical engineering and they have little or no knowledge of the use of electronic laboratory equipment. However, most of them have learned the basic design techniques for digital circuitry.

It is quite difficult for someone with no knowledge of laboratory instrumentation to implement a circuit design using standard integrated circuits. A computer system has been designed and implemented to help them in the construction, debugging and testing of a digital circuit.

2. Overview of CAILD System

CAILD is a Computer Assisted Instruction system for Logic circuit Debugging and testing. It is implemented on a PDP-9 computer system. (See Figure 3 for CAILD Block Diagram).

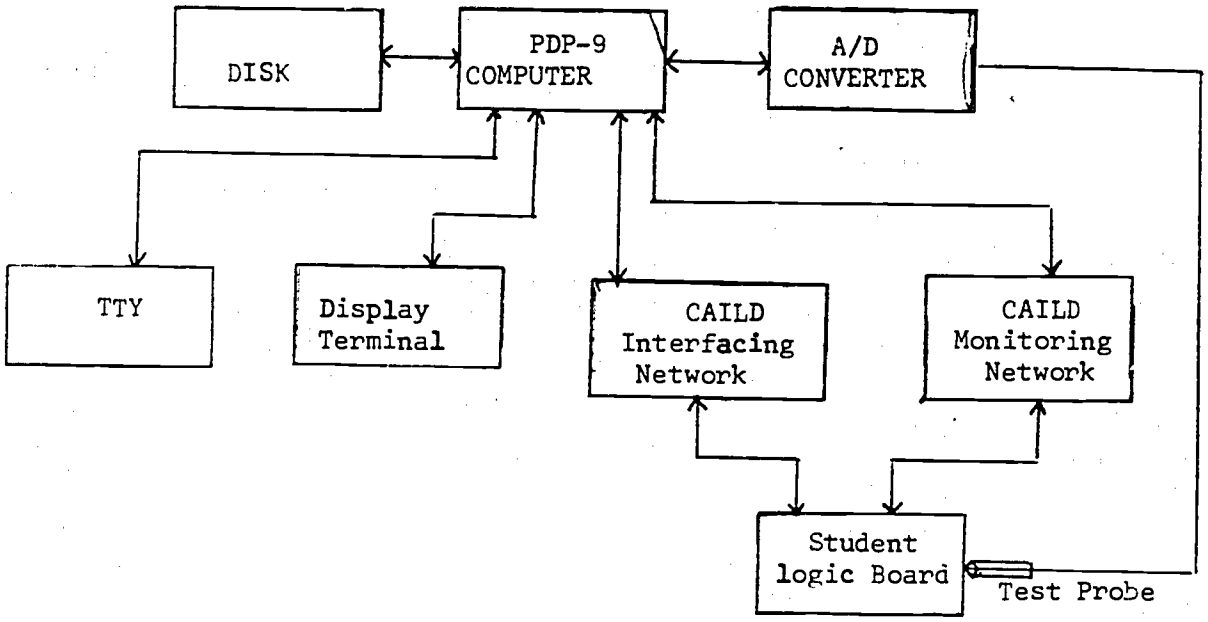


Figure 3: CAILD Hardward Configuration

By using the CAILD system, students are able to wire up their own logic network, debug it and test it without any knowledge of the use of laboratory instruments. The CAIL. system consists of a Student Logic circuit board, and interface network, a monitoring network, a test probe, a Tektronix 4010 display terminal and all systems software that is responsible for the instruction of logic circuit debugging and testing.

The student designs a circuit and prepares a wiring diagram prior to the use of the system. The student's circuit is wired on the Student Logic Circuit Board. NAND gates and flip-flops are provided on the circuit board. The student merely has to interconnect logic elements on the circuit board to build the designed circuit. The circuit board is then interfaced to the PDP-9 computer and the student inputs the circuit equations into the computer.

A syntax checker checks for syntax errors made in the equations and the NAND format conversion routine converts them into NAND format since only NAND gates are available on the circuit board. The debugger in the system compares the output of the physical circuit with the simulated output of the equations for each possible input condition in order to verify whether the physical circuit, in fact, realizes the given set of equations. If they do not agree, the system will guide the student through the whole debugging process until all the sources of error are found and corrected. See Table 10 for a sample session.

TABLE 10: CAILD Dialogue

```
      What is the output circuit point for variable J1?
** Q5
      Equation under test:
      J1=(-X1↑-X2↑Y3)↑(-X1↑X2↑-Y3)↑(X1↑-X2↑-Y3)↑(X1↑X2↑Y3);
      Do you wish to print out all test conditions
      (Y or N)
** N;
      Simulated output condition: 1
      Actual circuit output condition: 0
      Input conditions:  X1 X2 Y3
                         1  0  0
      What is the output circuit point for the following term:
      (-X1↑-X2↑Y3)
** P22;
      Simulated output condition: 1
      Actual circuit output condition:  1
      Input conditions:  X1  X2  Y3
                         1   0   0
      put the test probe on the input of the next higher level gate which
```

Table 10 - continued

```
      is connected to the output of (-X1↑-X2↑Y3)
      Type alt mode when you are ready.
** (User puts the test probe at the test point and types alt mode)
      Either you have a wiring error or an open circuit at the test point.
      Please correct the fault.  After the correction, type alt mode
      to return to the debugging phase.

** (User corrects the fault and types alt mode)
      Equation under test:
      J1=(-X1 -X2 Y3) (-X1↑X2↑-Y3)↑(X1↑-X2↑-Y3)↑(X1↑X2↑Y3);
      Do you wish to print out all the test conditions?
      (Y or N)
** N;
      Testing completed for one equation.
      Please tell me the circuit point for variable K1.
** 034;
                                  :
                                  :
```

Note: ** indicates the response for the user

After debugging the circuit, a test mode is entered. Under this mode the student is able to specify the input conditions and present state for a sequential logic circuit. If the circuit is a combinational circuit, the student specifies input conditions only. The system applies these student specified conditions to the physical circuit and returns the resulting output and next state to the student. By observing the output and next state of the physical circuit, the student is able to determine whether the physical circuit realizes the original design.

CAILD by itself is not a complete CAI system since it does not teach how to design logic networks. However, CAILD can be used in conjunction with a CAI system in which students learn the basic concepts of computer science. Certain concepts in this system are devoted to the design of combinational and sequential logic circuits. The design problems are generated by this system. The solution to the design problem results in a set of equations which describe the circuit. At this point the student can wire up the circuit on the Student Logic Circuit Board. The circuit will be debugged and tested under the guidance of the CAILD system. Finally, the student will obtain a working circuit for the original design problem.

Students can build either combinational or sequential circuits. There are two sets of four flip-flops (JK and Delay) available, but only one set can be used at a time. Hence, sequential circuits with up to sixteen states can be designed.

There are two external input lines, 32 NAND gates and twelve inverters. The outputs of either set of flip-flops can also be used as external circuit inputs for combinational circuits. This allows the design of fairly sizeable combinational circuits involving up to six input lines. The number of outputs for combinational circuits is limited only by the available logic elements.

3. System Evaluation

CAILD has been used in an introductory computer science course which teaches the design of digital networks as well as programming a minicomputer. There were twenty students in this class. Eleven were electrical engineers; the rest were predominately mathematics majors. None of the students had any prior exposure to digital design. The electrical engineers were taking their first electronics laboratory and would encounter experiments on digital logic design later on in the semester. The majority of students were sophomores.

The students in this class were required to design, construct, and debug both a combinational and sequential circuit with the aid of CAILD. The majority of students were able to successfully complete this task in about three and one-half hours (≈60 minutes design, ≈90 minutes wiring, ≈60 minutes debugging and testing with CAILD). Some of the circuits designed were full-adders, code-converters, decimal counters, and shift registers.

All but two of the non-electrical engineers found this to be a very stimulating part of the course as it gave them some hands-on experience applying the concepts they had learned in class. Similarly, all but two of the electrical engineers found this to be a very helpful preparation for their future project in the electronics laboratory.

The basic system is currently being expanded to include additional subroutines which will enable the student to make use of CAILD prior to actual circuit construction. Students will be able to verify that their equations accurately represent the transition table or truth table they had in mind. A wiring diagram will be printed out and saved by CAILD for use during the debugging process.

In summary, we believe CAILD is an effective tool for teaching the use of integrated circuit chips. It also teaches students how to locate faulty components and detect wiring errors. It makes their classroom work more meaningful as it provides them with some actual design experience.

The application to digital network design is, of course, most natural. However, the basic philosophy of CAILD could be used in the computerized teaching of other laboratory courses.

IV. A GENERATIVE APPROACH TO THE STUDY OF PROBLEMS

1. Introduction

This portion of the report describes research on problem solving for generative CAI. The goal of this research is a formalism for problem generation which is adaptable to many different subject areas. Since any implementation for a specific problem area will make use of application dependent information and procedures, the general formalism is intended to provide an approach which can be tailored and extended as application requires.

Following sections include a model for generation, a formalism which provides some details of the model, and examples which illustrate the use of the model.

The model incorporates heuristics which correspond in a dual manner to Polya type heuristics for problem solution. The intent of the generation paradigm might well be labeled, how to generate it. The formalism studies the possible structure a problem can have in terms of basic problems and the relationship of this structure to that of the solution to the problem. Possible structures of a problem are expressed as operations on subproblems.

Heuristics of Problem Solving

In this study we should not neglect any sort of problem and should find out common features in the way of handling all sorts of problems; we should aim at general features, independent of the subject matter of the problem. Moreover, "the study of heuristic has 'practical' aims; a better understanding of the mental operations typically useful in solving problems" will be directly applicable to the teaching and learning process.

In How to Solve It (12) Polya describes a problem as consisting of three parts: the unknown, data, and condition. He presents corresponding heuristics which can help to find a solution to the problem. Pólya presents a general approach to solving a problem: 1) understand the problem 2) devise a plan 3) carry out the plan 4) examine the solution.

A problem can be understood by identifying its parts (unknown, data, and condition) and by-establishing relationships among these parts with respect to a base of semantic information.

A plan for solution may be obtained by using the semantic base to find a connection between the data and the unknown,subject to the restraints specified by the condition portion of the problem. The plan will make use of subgoals which can be derived by using related problems. A subgoal may be part of the original problem, a similar problem having the same unknown, the same problem with auxiliary

elements introduced or terms replaced by their definitions. Problems may be related according to subpart, analogy, specialization, or generalization. Related problems can be arrived at by varying and modifying the three parts of a problem. These heuristic guidelines are more of an 'art' than a science; it is an art to recognize the utility of a related problem.

A plan consists of a sequence of subproblems and subgoals, each of which should be solvable, such that the combined solutions give a solution to the original problem. A plan directs the synthesis of a solution routine out of subgoal solutions.

The third step is the implementation of the plan. Each subgoal of the plan must be attained or an alternate found. A plan is a skeleton of a procedure for obtaining the solution. A plan must be refined from a general form to a detailed procedure. During this progression it may turn out that the plan fails. If so it must be patched or discarded.

The last step is solution verification. The synthesized solution routine can be checked by specialization, generalization, or by variation of the data. Specialization is the application of the solution routine to a special case or subclass of problems; generalization, the application to a more general problem. The solution should work for special cases and for varied data values and perhaps may work for other problems.

The four steps of 1) understanding 2) planning 3) execution 4) verification are themselves a general plan for a general problem solver.

2. The Generation Process

Generation is the process of producing many specific items from a general object. Generation is dual ('reverse' process) to analysis which is the passage from the specific to the general. The specific items are called interpretations; the general object, a representation.

Formally, the relationship of a representation to its interpretations is defined as a set function from the collection of representations to the collection of classes of interpretations. This set function and a corresponding representation have the same expressive power as the collection of all its interpretations, and, moreover, offer the potential of more powerful operations and transformations.

By making the set function dependent on other variables it can be constrained to map into a particular subclass of interpretations. For example, it may be made dependent on student performance or the modality of a verb; the former, for the generation of problems in CAI and the latter, for the generation of sentences having verbs of a desired modality.

CAI, and, in particular, the teaching of problem solving, (4,13) provided the initial motivation for this investigation. Koffman utilized a concept tree, (13), the arcs of which corresponded to subproblem or prerequisite relationships, as a control structure which determined possible calling sequences among the generation and solution routines of his CAI system.

A. Problem Generation

For problem generation the representation involves a set of objects and possible relationships which hold between them. The set of objects and relationships is called a problem model. A problem itself is an inquiry which seeks properties or consequences of a given interpretation of the model.

Following Polya, problems have a goal or solution, which may be an unknown value, a sequence of actions, a program, a proof, etc. In addition, all problems have explicit information, the data, which is given and is usually needed to solve a problem. Other information which is implicit may also be needed to solve the problem. This implicit information is not part of the data, it may be contained in a semantic network. Finally, the goal or solution is determined by the condition expressed and any relationships implied by the problem model.

This paper will deal with quantitative problems as opposed to problems of non-numeric reasoning, and the representation used will be a LISP list of the form ((unknown variables)(data variables)(relation)) where each of the three items in the list is a sublist and (relation) is a LISP routine which expresses a relationship which holds among the data objects and the unknowns.

Denote a problem representation by P_i. Let C_i denote the class of problems represented by P_i. Problem generation, then, is a map which has P_i as a dependent variable and which takes values in C_i.

The sublists determine problem subclasses. For example, let C_i be the collection of force, mass, acceleration problems of elementary physics. Than a subclass of C_i is represented by P_{ij} = ((force)(mass acceleration)(function)) where function expresses the relationship given by the equation force = mass · acceleration.

In this context, an implementation of a generation map can involve several phases. In the first phase, the sublists can be manipulated, combined, or portions of them generated or deleted to produce new lists. A second phase could provide a semantic interpretation of the representation list, using, for example, a semantic network. A third phase could then express the interpreted list in a

form natural to a user.

The first phase is of primary concern here because the generation of representations will provide a significant increase in generative capability.

3. Duality and Problem Solution

The solution to a problem is essentially determined by the expressed and implied relationships which hold between the data and the unknown items of a problem. Indeed, the discovery of these relationships is often the goal in solving a problem. For example, the function of the representation ((force)(mass acceleration)(function)), gives the key to the subclass of force, mass, acceleration problems represented.

Thus, in the list representation, the problem itself is represented by ((unknown)(data)) and the solution by (function).

Let C be the class of problems under consideration. A problem solving system is described by a pair (R, S) where R represents a subclass of C and S is a solution method or routine for this subclass.

Given a collection $\{(R_i, S_i)\}$, the formalism below investigates the following set of interrelated questions:

1) how can the problem subclasses be extended while keeping the collection of solution routines, $\{S_i\}$, the same
2) what are some relations on $\{(R_i, S_i)\}$; what structures exist on this collection
3) what manipulations and operations can be performed on the lists P_i to produce new representations
4) (duality) what are corresponding operations on the procedures S_i which produce solution routines for the new classes.

The duality of problem and solution operations will be expressed by an operator pair which operates on the list representations, i.e. the problem operator acts on ((unknown)(data)) and the solution operator part acts on (function).

The objects (R_i, S_i) are general and S_i could represent special routines, such as (DEFUN SFORCE (X Y) (SETQ FORCE '(PRODUCT X Y)) where X will be mass and Y acceleration. S_i could also be a general problem solver such as a theorem prover. An appealing use of a general deductive system (such as a theorem prover) would be for high level inferences for the manipulation of the (R_i, S_i). Also, the determination of which S_i to employ and how to use it to solve a problem would be decided by some other S_i'.

An example of a relationship which holds among problems is that of sub-problem. The dual relationship on the solution routines is subsolution. This structure is rather simple and obvious. Nonetheless, it is important for a system to have a way of representing knowledge of such structures. What are other structures? The structural information which a generative system uses is knowledge that a problem solving system should also have.

4. Abstract Problems

This following section illustrates the approach to problem generation and solution which is used to find answers to the above questions.

Certain subclasses of a given collection of problems are represented by a structure called a basic abstract problem. Some relations on abstract problems are introduced and a partial order on them can be defined. A complex abstract problem is one that is constructed from several basic abstract problems. Considerations of composition techniques using the relations are made.

To each basic abstract problem there will correspond a problem solver, or solution method. Corresponding relationships for solution methods are also considered. A complex abstract problem will be solvable using a combination of the problem solvers associated with its basic abstract problem constituents.

Definitions and Elementary Results

An abstract problem is defined to be a triple of the form ((unknown)(data)(relation)) where (unknown) is a list of variables whose values are sought as the solution to a problem represented, (data) is a list of input variables, and (relation) defines a function which assigns unique values to the unknowns for each list of data values.

An abstract problem which is input to the first phase of problem generation is said to be basic. Basic abstract problems are assumed to be solvable, that is have a given associated solution routine.

For example, ((FORCE) (MASS ACCELERATION) (SFORCE MASS ACCELERATION)) is a basic abstract problem.

An interpretation (unevaluated) of an abstract problem is defined to be an association of properties, objects, relationships to the data and the unknown. The associations must be semantically meaningful (for example, according to a semantic network) with respect to the sublists of the problem representation.

For example, an interpretation of the abstract problem given above might be the association of a physical object with properties of mass, acceleration, and another physical object with the property force, such that the second object exerts its force on the first object. A problem in the class represented might be: a ball has a mass of x pounds and an acceleration of y feet per second. What force was exerted on it when it was hit by a bat?

The predicate of an abstract problem ((unknown)(data)(function)) is the predicate PR such that PR (unknown data) is true if and only if the value(s) of the unknown(R) are the value(s) given by the function defined by (function) when evaluated on (data). Predicates can be used for formal proofs concerning abstract problems. If the predicate of a problem can be proved from the predicates of basic abstract problems, the structure of the solution in terms of basic solution procedures is determined by the proof.

A. Constructions on Abstract Problems

The constructions on abstract problems involve set theoretic and functional operations. Constructions include subproblem, specialization, generalization, analogy, transformation, union (or concatenation), composition, cascade, and domain dependent constructions. These constructions are implemented as operators which apply, on the one hand, to sublists R_i and, on the other, to solution routines S_i. The main syntactic generative operations are union, composition, cascade, and certain kinds of transformations. Subproblem is a decomposition technique which yields, ultimately, basic problems. Specialization, generalization, and analogy are also special cases of transformation. They have a syntactic aspect but also involve unsolved semantic issues.

B. Power

A problem generation and solving system is described by a triple (C R S) where R is a representation for the class of problems C which can be solved by the solution method S.

The size and variety of C is one measure of the power or capability of such a system. Power can be increased by enlarging C while keeping S fixed 2) enlarging C while extending S or by 3) decreasing the degree of specialization required of the user to implement S while keeping C and S (the method) fixed.

The following discussion is primarily directed towards 1) and 3). For example, let $(\{C_i\},\{R_i\},\{S_i\})$ be a collection where C_i is a subclass of problems represnted by R_i and solvable by S_i and the collection is part of a CAI system. C_i deals with a particular concept of the course, R_i may be a generative grammar or a generation routine which generates problems dealing with the concept, and S_i solves the generated problems and monitors the student solutions. The system can be extended by simply adding more basic elements, (C_i, R_i, S_i). Relations on the basic elements could be defined. This introduces a structure on the collection of basic elements, which can be represented by a graph, called a concept graph in a CAI application (13). The intent of the relations is to increase the size of $C=U_iC_i$ or to decrease the degree of specification required to express solution routines. The relations, in effect, serve as part of the control structure. Koffman used the relations of subproblem and prerequisite problem and these determined possible calling sequences among the routines $\{R_i\}$ and $\{S_i\}$. The relations will serve as part of the definition of a generation map so that a set of elements (or abstract problems) can be represented and replaced by another smaller set of basic elements and a generation map. Finally, the system could be made more powerful by giving it the capabilities of determining the structures on the basic elements or of determining when the relations hold rather than having the structures explicitly stored or indicated, say by a graph.

The basic abstract problems are those which will be used as building blocks out of which complex abstract problems, not explicitly represented, can be constructed.

Further, the basic abstract problems are assumed to be solvable, and, depending on the constructions used, the complex abstract problems will also be solvable by an appropriate combination of the basic solution routines. We will be interested in studying the effect of problem operations on problem generation and problem solution routines.

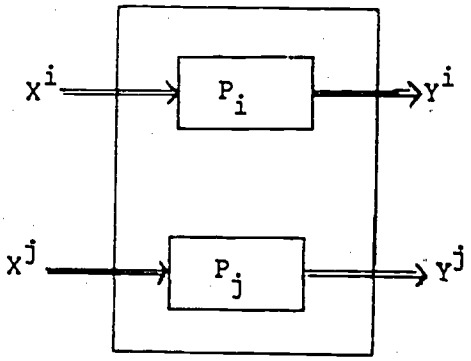
C. Union

Let P_i and P_j be two abstract problems with $P_i = ((Y^i)(X^i)(S_i))$ and $P_j = ((Y^j)(X^j)(S_j))$ with predicates PR_i and PR_j, respectively.

The union or concatenation of P_i and P_j, denoted $P_i \cup P_j$, is defined to be $(((Y^i)\cup(Y^j))\ ((X^i)\cup(X^j))\ (S_{P_i \cup P_j}))$ where $S_{P_i \cup P_j}$ is a solution routine for all

the problems represented and is determined by the conjunction of PR_i and PR_j. 'U', union, on the variables indicates set union. The superscript denotes a list of variables.

The union of problems is depicted by the diagram



P_i U P_j

The implementation of union of problems is denoted by PUNION.

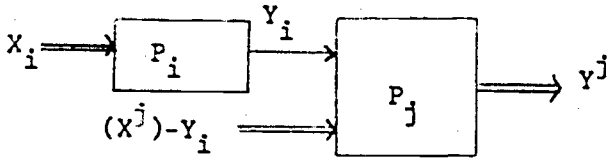
D. Composition

Another syntactic type of operation is composition. Composition makes use of substitution, which, also, is at the 'heart' of resolution. Moreover, composition is fundamental to LISP.

Suppose $P_i = ((Y_i)(X^i)(S_i))$ and $P_j = ((Y^j)(X^j)(S_j))$ where Y_i is a single element and $(Y_i) \subseteq (X^j)$.

$P_j \bullet P_i$, the composition of P_i and P_j, is defined to be $((Y^j)((X^i)$ U $((X^j)-(Y_i)))$ $(S_{P_j \cdot P_i}))$ where $S_{P_j \cdot P_i}$ is determined by the predicate $PR_i((Y_i)$ $(X^i))$ $\wedge$ $PR_j((Y^j)((Y_i)$ U $((X^j)-(Y_i))))$.

Diagrammatically, $P_j \cdot P_i$ is given by



The implementation of composition is called PCOMPOS.

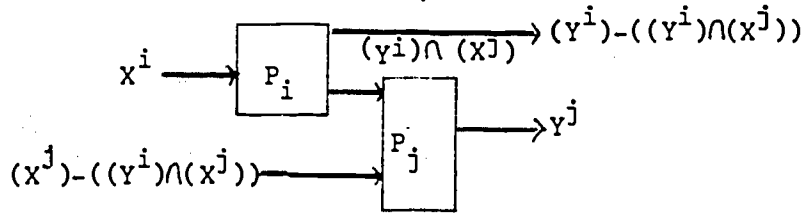
E. Cascade

The cascade operation is a generalization of both the union and composition operations.

Again let $P_i = ((Y^i)(X^i)(S_i))$ and $P_j = ((Y^j)(X^j)(S_j))$. Also, assume that $(Y^i) \cap (X^j)$ is not empty.

P_j o/o P_i, the cascade of P_i and P_j, is defined to be $(((Y^j) \cup ((Y^i) - ((Y^i) \cap (X^j)))) ((X^i) \cup ((X^j) - ((Y^i) \cap (X^j)))) (S_{P_j \text{ o/o } P_i}))$, where $S_{P_j \text{ o/o } P_i}$ is determined by the predicate $PR_i ((Y^i)(X^i)) \wedge PR_j((Y^j)(((Y^i) \cap (X^j)) \cup ((X^j) - ((Y^i) \cap (X^j)))))$.

The cascade of P_i and P_j is illustrated by the following diagram



The implementation of cascade is denoted PCASCADE.

F. Map

Another natural relation which can hold among abstract problems is that of map or transformation. Let P_i and P_j be as above.

A map from P_i to P_j is a pair (f_1, f_2) where f_1: $(Y^i) \longrightarrow (Y^j)$ and f_2: $(X^i) \longrightarrow (X^j)$ such that if $PR_i((Y^i)(X^i))$ is true then $PR_j(f_1((Y^i)) f_2((X^i)))$ is also true.

Then, if P_i is an abstract problem and $f=(f_1, f_2)$ is a pair of functions such that $PR_i((Y^i)(X^i))$ true implies that $PR_i(f_1((Y^i)) f_2((X^i)))$ is true, then the construction by map gives the abstract problem $f(P_i) = ((f_1((Y^i)))(f_2((X^i)))(S_i))$.

If the condition on the predicates does not hold it will be necessary to modify S_i to obtain a new solution routine.

It is possible to define equivalence and partial orders of abstract problems and these could be shown to be preserved by a map of abstract problems.

A simple example of a map for elementary physics is one which changes the units of measure.

The definition of a higher order map would include changes to the solution routine in addition to the variables. Thus, a higher order map would be a triple, (f_1, f_2, f_3), where f_1 and f_2 are as above and f_3 modifies S_i such that $f_3(S_i)$ assigns the proper values to $f_1((Y^i))$ for data values of $f_2((X^i))$.

Some special types of maps are analogy, specialization and generalization.

An implementation of a map will be labeled TRANS.

5. Solution Routines

Solution routines for problems resulting from the above constructions are determined by predicates obtained from corresponding logical operations on the predicates of the operand problems.

The solution routines for constructed problems can, therefore, be obtained from solution operators which correspond to the respective logical operations. These solution operators operate on basic solution routines to synthesize complex solution routines.

Implementations of the solution operators for PUNION, PCOMPOS, AND PCASCADE will be termed SUNION, SCOMPOS, AND SCASCADE, respectively.

Using this notation union of abstract problems, for example, becomes P_i U P_j= $(((Y^i)U(Y^j))\ ((X^i)U(X^j))$ (SUNION S_i S_j)).

(SCOMPOS (A) (B) (S)) will mean to substitute A for B in the routine S. A similar notation will be used for SCASCADE.

Propositions:

It is easily seen that PUNION is associative and commutative. PCOMPOS and PCASCADE are associative.

Also, if P_j is a union of abstract problems, then $P_j \circ P_i$ can be written as a union, that is o is left distributive over union.

Using the basic abstract problems and the operations of union and composition, complex problems can be constructed. By the left distributivity of composition over union each of these complex problems can be written in a standard form.

For example, (PCOMPOS P_1 (PUNION P_2 (PCOMPOS P_3 (PUNION P_4 P_5)))) = (PUNION (PCOMPOS P_1 P_2) (PUNION (PCOMPOS P_1 (PCOMPOS P_3 P_4)) (PCOMPOS P_1

(PCOMPOS P_3 P_5)))). If PUNION is defined as an n-ary operation (a FEXPR in LISP), the expression becomes (PUNION (PCOMPOS P_1 P_2) (PCOMPOS P_1 (PCOMPOS P_3 P_4)) (PCOMPOS P_1 (PCOMPOS P_3 P_5)))), which has the form (PUNION $P_a P_b P_c$ ...) where each operand is a basic abstract problem. Similar computations hold for the solution operators.

These results are used to implement SCASCADE.

Intuitively, SUNION concatenates its argument routines. SCOMPOS makes a substitution for a variable in a given routine. (SCASCADE (A) (B) (C)) means to substitute each element of the list (A) for a corresponding variable in the list (B) for each occurrence of that variable in the routine C. Then SCASCADE can be implemented by first putting the complex solution of P_i (in P_j o/o P_i) in standard form and repeatedly applying SCOMPOS until all the substitutions have been made.

6. Control

The control which specifies the manner in which abstract problems are combined and how a solution routine is synthesized is contained within the operators themselves. The decision when to apply the operators and which problem operands to be operated on is made by a planning routine. For some applications, the planning routine need only consist of a random selection mechanism for selecting operators and problem representations to be operated on. For instance, for problem generation for CAI, such decisions could be made randomly, with constraint by a complexity measure.

For other applications and other operators, the planning routine will need semantic information. In these situations the planning routine could incorporate path tracing and pattern matching mechanisms for tracing within a semantic network.

There is an analogue with programming languages. Here the concern is that of combining lists and procedures using operators via a planning routine rather than combining elementary statements via a program. Also, here the attempt is to incorporate decision making capability within the planning routine.

7. Computational Issues

Given n basic abstract problems, the number of meaningful abstract problems which can be generated using PUNION, PCOMPOS, AND PCASCADE can easily be seen to be bounded below by n and above by 2^n-1.

This does not say that the number of meaningful representations generated will approach the upper bound. The actual number which can be generated depends on the mutual relevance of the given n basic representations. For CAI applications

most useful problems will be the result of at most 3 or 4 applications of the operators.

On the other hand recall that the discussion still pertains to the representation level and each representation will be used in conjunction with, say, a semantic net to produce many specific problems.

Another example

The above operators are natural ones for quantitative subjects and for use in LISP.

Suppose a list of basic abstract problems from the topic of uniformly acceleration motion consists of the following three representations:

1) ((DISTANCE) (AV-VEL TIME) (SDIS AV-VEL TIME)) where distance, average velocity, and time are written as LISP atoms which point to a semantic net and SDIS is a LISP function which returns a value for distance given values for average velocity and time.

2) ((ACC) (VEL-INIT-VEL TIME) (SACC VEL INIT-VEL TIME)) where SACC returns a value for ACC given velocity, initial velocity, and time as arguments.

3) ((AV-VEL) (VEL INIT-VEL) (SAVVEL VEL INIT-VEL)) where SAVVEL is a LISP routine which defines average velocity.

PCOMOS applies to 3) and 1) gives ((DISTANCE) (VEL INIT-VEL TIME) (SCOMPOS (SAVVEL VEL INIT-VEL) (AV-VEL) (SDIS AV-VEL TIME))), where SCOMPOS substitutes the expression (SAVVEL VEL INIT-VEL) for AV-VEL in the function SDIS before it is evaluated.

Transforming 2) and composing with the representation just generated would give ((DISTANCE)(INIT-VEL TIME ACC) (SCOMPOS (TRAN VEL (SACC VEL INIT-VEL TIME)) (VEL) (SCOMPOS (SAVVEL VEL INIT-VEL) (AV-VEL) (SDIS AV -VEL TIME)))).

TRAN would be a symbolic manipulation routine for solving for velocity in terms of acceleration, initial velocity, and time given (SACC VEL INIT-VEL TIME).

The resulting abstract problem has a solution routine which corresponds to a derivation of the equation distance = (initial-velocity · time) + (acceleration · $time^2$)/2.

8. Conclusions

The paper has represented a model and approach to problem generation and solution, which consists of an expression duality between problem generation and problem solution, a study of the structure of a problem in terms of basic problems, the expression of structural relations as operators and routines, the elevation

of some control to a higher level (to a structure on abstract problems and solution routines rather than one on elementary programming statements). This approach seems to allow easy incorporation of direction and semantic knowledge, large inference, steps, and additional inference operators. It is related to formal logic via the predicate of an abstract problem and has significance for automatic synthesis of programs because it discusses the synthesis of routines out of routines rather than elementary statements; for algebraic manipulation, since it suggests a flexible notation and approach with the possibility of knowledge manipulation.

The model and development provide a context or setting which brings forth many additional questions such as the incorporation of static knowledge to extend some of the operators beyond their formal manipulation capability, questions on the interplay of static knowledge in a data base or net and dynamic knowledge in the form of routines, 'surgery' for the modification of a routine, and applications to other problem areas including programming languages.

V. A NATURAL LANGUAGE LISP TUTOR

1. Introduction

Natural language understanding is becoming more important in CAI systems. Many powerful parsers have been developed for very extensive subsets of English including Wood's parser (14) and Winograd's PROGRAMMAR (15). These methods are beginning to be of use for question answering sub-systems and for interpreting student responses. Of particular interest is the work done by Brown et al. (16). It is his semantically driven fuzzy grammar that has influenced this research.

Brown analyzed student scenarios involving electronic circuit trouble-shooting and discovered that the concepts and items discussed fell into small numbers of semantic groupings or categories.

These included: measurements, control settings, circuit elements, and circuit locations relative to the electronic circuit. The parse was then accomplished by anticipating and looking for phrases which matched these semantic categories where they might be appropriate. The parser's fuzziness stems from the fact that words that are not understood can be skipped over.

Teaching a programming language embraces two major tasks. One is presenting new material in an orderly controlled fashion. The other involves making available a friendly, informative environment that allows experimentation, and that can recognize errors and misconceptions and point them out. With ISLE (Interactive Student LISP Environment) we have concentrated on this second item by developing a computer based environment for LISP. The system knows about the structure and semantics of the LISP language and can answer questions about LISP data structures, concepts and functions; both those functions defined by the student and the primitives internal to the system.

As an example consider the following dialogue:

STUDENT: Is Atom a functior.
MACHINE: Yes, Atom is a subr.
STUDENT: How many arguments does it take?
MACHINE: Atom is a subr and takes 1 argument.

Notice that questions are answered with a complete statement to avoid any ambiguity, also note the use of the pronoun it by the student. Some limited anaphoric reference is permitted by the system to allow a more natural dialogue.

```
STUDENT:  Describe it
MACHINE:  Atom is a subr. and a predicate function
          It returns true (the Atom T) if its argument
          is an atom, nil otherwise.

STUDENT:  What is an Atom?
MACHINE:  An Atom is the most elementary type of s-expression.
          It may be either a number or a literal atom.
```

Some words like atom can refer to more than one thing. The correct reference is found by looking at the context of the word and making a decision based on plural markers, determiners, or qualifying adjectives.

```
STUDENT:  Are property lists kinds of lists?
MACHINE:  A property or P list is an alternating list of literal
          atom property names and corresponding property values.
          It can be search using the function GET.
```

Subset-superset and part-part of relationships are used extensively by the system for answering questions of this type. A yes answer is implied here by giving the student a definition of the subset item.

```
STUDENT:  Define (((FACT (LAMBDA (X) (COND
                        ((ZEROO X) 1)
                        (T (TIMES X (FACT (SUB 1 X))))))))))

MACHINE:  Arguments for Evalquote:
          DEFINE
          (((FACT (LAMBDA (X) (COND((ZEROO X) L) (T(TIMES X
          (FACT (SUB1 X)))))))))
          VALUE IS:
          (FACT)
```

Any student input that has the form of a function followed by an argument list is assumed to be input for the evaluator. Here the student defines a function to compute factorials.

```
STUDENT:  FACT (3))
MACHINE:  Arguments for Evalquote:
          FACT
          (3)
          VALUE IS:
          Undefined function found.
          It is ZEROO found in the function fact.
```

In defining FACT the student mistyped ZEROP. The error is found during evaluation and returned to the student. Notice that terminating symbols are necessary at the end of each student input line. They may be periods, question marks or an extra right parenthesis.

```
STUDENT:  PRINT FACT.
MACHINE:  (LAMBDA (X) (COND & &))
```

When printing function definitions only the highest elements of the definition appear; lower level structures are replaced with &.

STUDENT: 3 2 p.
MACHINE: ((ZEROO X) 1)

Now using commands to the editor, the student looks at the lower level where he remembers ZEROP should go.

STUDENT: 1 (1 ZEROP) P.
MACHINE: (ZEROP X)

The error is now corrected and this can be checked by evaluation or by asking another question.

STUDENT: Does FACT call any undefined functions?
MACHINE: FACT calls no undefined functions.

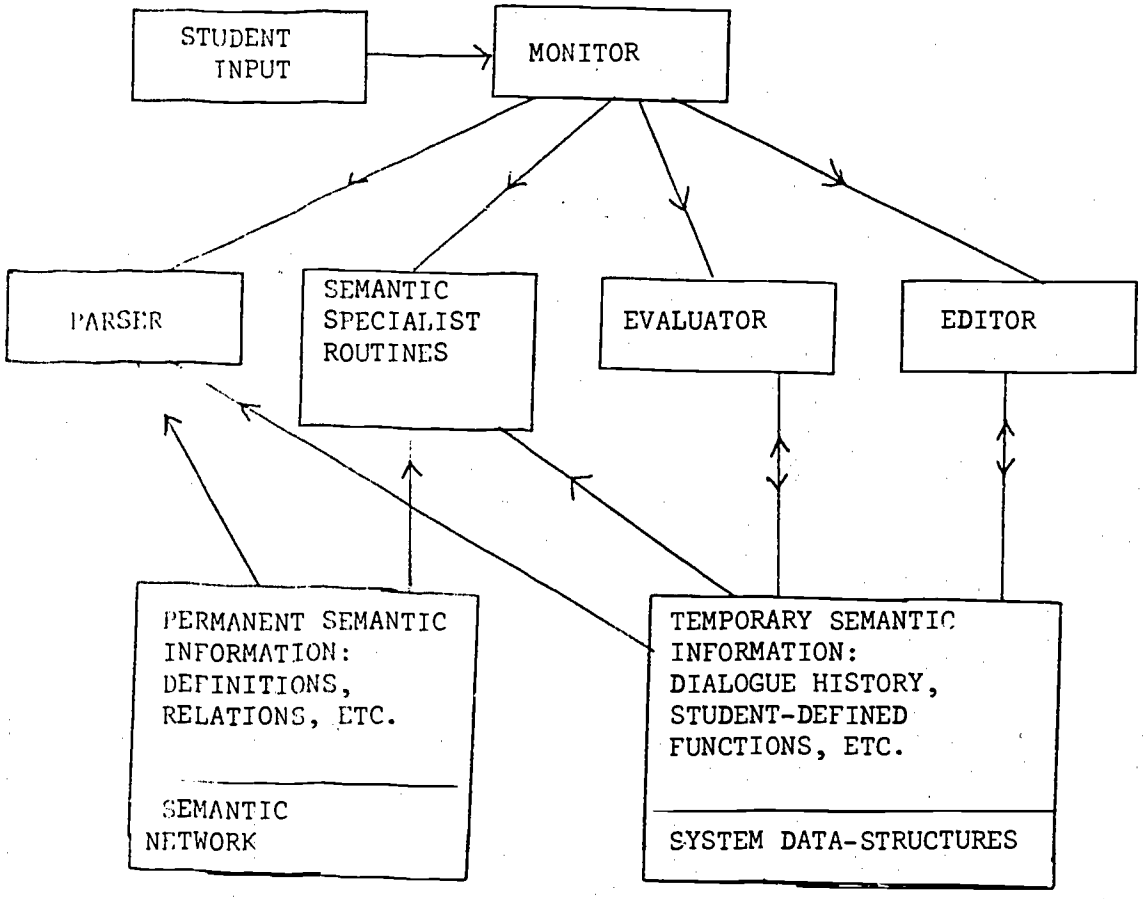
STUDENT: FACT (3))
MACHINE: Arguments for Evalquote.
FACT
(3)
VALUE IS:
6

System Organization

The system is built around several submodules as shown in Figure 4. Each Student command or request is received by the monitor where it is classified as input for either the editor, the evaluator, or the parser. Any input which is not either a list of edit commands or a function followed by an argument list is assumed to be a request for information and so is passed to the parser. Anything the parser can not handle is rejected and the student is asked to rephrase or break up the question.

The parser and the semantic routines work together to answer factual questions about the LISP language, its data structures and functions, and about the functions that student has defined. The parser mechanism has a semantic basis in that specific words or phrases are looked for that refer to things the system knows about, i.e. functions, definitions, or entries in the semantic networks. These items are divided into groups or categories that are semantically similar.

Semantically similar items are those that might fit in a given slot in a sentence or question, and that fall into a superset classification such as data



ISLE System Organization - Figure 4

structures or function names. The result of the parse is an executable LISP function whose evaluation causes a response to be generated for the student.

The evaluator evaluates student functions when called upon, accepting nearly any LISP 1.5 constructions. When student errors are found, it reports the type of error and in what function it occurred to the student. Editor commands can then be used to look around inside of function definitions and to insert, delete, and change parts of the definition.

THE GRAMMAR AND ITS IMPLEMENTATION

The heart of the English understanding component of the system is a BNF grammar. After a line has been read in, an interpretation of it is attempted using an implementation of the grammar shown in Appendix B. In the SOPHIE system [16], every non-terminal is considered a semantic entity to be searched for when necessary. In the ISLE system, however, only a few of the rules are actually concerned with semantic entities or categories. These semantic entities are defined as only those things which have entry in the semantic network. The rules which embody certain semantic groups have already been described. The rest of the grammar rules are used to identify requests for certain relationships or properties of the semantic entities.

Most programs which make use of a grammar use some kind of parser or grammar interpreter. This parser (a program) then uses a table or array in which the grammar rules are stored (data). Special control structures must be set up to control backing up when an incorrect parse is begun. In ISLE, this grammar is implemented directly in LISP. For each rule (non-terminal) in the grammar, there is a corresponding LISP function with the same name implementing that rule. The LISP control structures make this implementation relatively easy due to the recursive definition of LISP functions in general and the use of the special built-in

functions; COND, AND, and OR. Backup is automatic as each rule-function can let its calling rule-functions know of its failure on return. All pointers and variable values will again be those originally set in the calling function. There is nothing to undo or redo as the LISP control structure handles this automatically.

THE SEMANTIC ROUTINES

The parsing operation, if it is successful will produce another LISP function to be evaluated. Some of these functions and the sentences that produced them are given in TABLE 11. Each is a call to a predefined semantic routine. The functions FN, FTYPE, CONCEPT, and STRUCTURE retrieve the desired semantic information for their arguments. In this way words such as ATOM are disambiguated. For example, (FN ATOM) will retrieve information relevant to the function ATOM, while (STRUCTURE ATOM) will retrieve the information concerning the structure. PREF is used to find semantic information for pronouns which it does by matching its arguments against the semantic categories of previously mentioned items.

TABLE 11 Sentences and Their Translation Into LISP Functions

IS ATOM A FUNCTION?

(RELATE (FN ATOM) (FTYPE FUNCTION))

HOW MANY ARGUMENTS DOES IT TAKE?

(ARGCOUNT (LIST (PREF FN)))

DESCRIBE IT.

(DESCRIBE (LIST (PREF FN FTYPE CONCEPT STRUCURE)))

WHAT IS AN ATOM?

(DESCRIBE (LIST (STRUCTURE ATOM)))

ISLE's semantic routines are all specialists for answering their own types of questions. Some take information directly from the network to be given to the student or to be used in comparison or relationship tests. DESCRIBE, for example, gives the student a pre-defined definition or description if it exists. In the case of student defined functions, it tells the student the type of function it is. RELATE reports on 'superset' 'subset', and 'part-of' relationships between its arguments. ARGCOUNT checks the semantic information associated with its argument, or in the case of student defined functions-the actual function definition, to tell how many arguments a particular function has.

The permanent semantic information used by these functions is set up as association lists of relationships and values for each semantic entity. Table 12 shows this information for the structural item atom. The value of the relationship TEST is the name of a predicate function which tests for the associated semantic entity. In this case, the function ATOM tests for the structure which is an atom. TYPE and TYPE OF indicate subset and superset relationships, and DESCRIPTION indicates a literal definition of the item.

TABLE 12 Semantic Information for the Structure ATOM

```
((TEST . ATOM)
(TYPE OF . (S-EXPRESSION INDICATOR))
(TYPE . (LITERAL NUMBER))
(PART OF . (S-EXPRESSION DOTTED-PAIR LIST))
(DESCRIPTION . ((AN ATOM IS THE MOST ELEMENTARY TYPE OF
        S-EXPRESSION (DOT))
        (IT MAY BE EITHER A NUMBER OR A LITERAL ATOM (DOT)))))
```

Other temporary information that might be used by the semantic routines can be created and changed in various ways. When a student defines a function, the function is analyzed and lists of the variables it binds or uses and the functions it calls are created. This information is used by the routines which handle questions about the student's functions and is updated whenever a function is edited or redefined. The editor and the evaluator also store information that could be used by the question-answering system. This is done whenever errors

occur and includes information about the current state of the evaluator or editor (e.g. the association list) and the cause of the error. This would allow the student to obtain more information about the source of the error and what the evaluator (or editor) was doing before the error occurred.

CONCLUSION

This system is undergoing continued development. The question-answer is being expanded to allow the student to get more of the information he or she might want and to perform more edit functions in English.

ISLE is implemented in LISP which runs interactively on an IBM 360/65. This interactive LISP is an improved version of the Waterloo LISP which uses a cathode-ray display as the active user terminal.

Preliminary indications are that the system will serve as a useful tool for familiarizing a student with LISP concepts. The question answering capability allows a student to inquire about the semantics of LISP; he can use the LISP student evaluator to test his knowledge of LISP syntax and to help him correct his errors. The expanded diagnostic information presented should help him clear-up initial misconceptions and ease his transition from ISLE to the standard LISP evaluator.

This approach appears to be general in that one could present any material of a factual nature in a similar manner. SOPHIE [16] is an example of a similar system for teaching electronic-circuit analysis and trouble-shooting. Other programming languages, logic circuit design, and basic algebra and calculus might possibly be taught using a similar computer environment.

VI OVERALL EVALUATION

This project has studied the topics of student modelling, concept selection, generative CAI, problem generation and solution, and natural language in CAI. As a vehicle for the study of these topics, several experimental teaching systems have been designed and implemented.

An introductory course in digital design and programming which utilizes the systems described in Sections I, II and III of this report has been taught to five sections of students. (approximately twenty-five students/section). The results of this experiment have been described above and were generally very satisfactory.

The conclusion from this experiment is that generative CAI is a very effective medium for teaching quantitative college-level courses. The major advantages are the guided direction in the problem-solving process provided to beginning students, and the presentation of instantaneous feedback and remedials when the student goes astray. Generative CAI also frees the instructor from discussing routine, algorithmic procedures in class, and allows him to concentrate on more complex concepts. Through the use of a monitor which makes "intelligent" decisions concerning the concept to be studied, the difficulty of the problem to be generated, and the degree of explanation and student monitoring provided; instruction can become highly individualized. In this way, the system makes the maximum use of each computer session. The student is also free to proceed at his own pace and override any of the monitor's decisions.

The major drawback to generative CAI is the cost of designing and operating these systems. The design was, of course, a one-time expense and entailed many man-hours of effort. Unfortunately, there is no accurate accounting available of the time spent in the implementation of this system. Many routines were written by the principal investigator; some were implemented as portions of Master's Thesis; others were independent study projects.

The cost of operating the system averages out to approximately $100 per student per semester. This provides on the average of four terminal hours per student per week. Certainly, this is expensive compared to the design goal (not yet attained) of the PLATO system of $.50 per terminal hour per student. Unlike PLATO, it should be noted that this system was not implemented on hardware configured and intended exclusively for CAI; rather, it was implemented on the IBM 360/65 computer which was not really designed for time-sharing.

To provide the same degree of individualized instruction for twenty-five students would require hiring five graduate assistants (assuming a 20 hour/week workload). The cost of five graduate assistants for a semester would be approximately $8,000. The cost of generative CAI (approximately $3500) would be considerably less.

Additional studies in the use of Artificial Intelligence techniques have provided a general model for the generation and solution of problems (Section IV). This model also appears to have applicability in areas of artificial Intelligence research such as Problem Solving and Program Synthesis.

A hardware-software system was constructed for use in the digital laboratory. This system was very successful in teaching students how to apply classroom concepts in the design of actual digital circuits.

In addition to the system described in Section III, a digital services system was built which acted as a "front-end" for the original system. The digital-services system aided students in all phases of the design problem and subsequently transferred control to the debugging phase of the original system to verify the correctness of the student's design. Any hardware bugs or faults were located, with student assistance, and eliminated.

A model for a natural language CAI system previously proposed by Brown (16) was adapted for use in an Interactive Student-oriented LISP Environment (ISLE). The implementation of this system was relatively straight-forward once the LISP interpreter was modified to run interactively on the IBM 360/65. Its use verified that AI research in natural-language processing could be effectively applied in the design of teaching systems.

One interesting feature of all of the research decribed herein is the broad spectrum of teaching activities and techniques covered. The activities range from electrical engineering courses in the theory, design, and laboratory construction of digital circuits to computer science courses in machine-language programming and the LISP computer language. Contributions were also made to problem generation and solution in general as well as problem-generation in high school algebra (5). The computerized teaching techniques included generative CAI, which was normally under system control, a digital circuit debugger which provided a mixture of student and system control, and finally a LISP learning environment which was entirely under student control

This research is a small step in the application of Artificial Intelligence techniques to CAI. It is expected that future research along these lines will contribute to the continued growth of Artificial Intelligence and provide significant enhancement to the intelligent use of computers in instruction.

REFERENCES

1. Bobrow, D., "A Question-Answering System for High School Algebra Word Problems," in Proc. AFIPS Fall Joint Computer Conference, pp. 591-614, 1964.

2. Winograd, T., "Understanding Natural Language," New York, Academic Press, 1972.

3. Nilsson, N., "Problem-Solving Methods in Artificial Intelligence," McGraw-Hill, 1971.

4. Koffman, E. B., "Design Techniques for Generative Computer-Assisted Instructional Systems," IEEE Transactions on Education, Vol. E-16, No. 4, pp. 182-189, November, 1973.

5. Gilkey, T. J., and Koffman, E. B., "Generative CAI in High School Algebra," Proceedings of the International Computing Symposium 1973, pp. 489-493, North Holland Publishing Co., 1974.

6. Koffman, E. B., amd Blount, S. E., "A Modular System for Generative CAI in Machine Language Programming," IEEE Transactions on Systems, Man and Cybernetics, SMC-4, No. 4, pp. 335-343, July, 1974.

7. Samuel, A., "Some Studies in Machine Learning Using the Game of Checkers," IBM J. Research and Development, Vol. 3, No. 3, pp. 211-229, 1959.

8. Walrath, S. J., "A Heuristic Monitor for Individualizing Instruction in CAI," unpublished M. S. Thesis, Electrical Engineering and Computer Science Dept., University of Connecticut, Storrs, Ct. 1974.

9. IBM Corporation, "Conversational Programming System (CPS) Terminal User's Manual," IBM Report, GH-20-0758-0, 1970.

10. Koffman, E. B., and Blount, S. E., "Artificial Intelligence and Automatic Programming in CAI," Proceedings of the 3rd International Joint Conference on Artificial Intelligence, Stamford, Calif. 1973.

11. Neal, J. P. and Meller, D. V., "Computer-Guided Experimentation - A New System for Laboratory Instuction," IEEE Transactions on Education, Vol. E-15, No. 3, August, 1972.

12. Polya, G. How to Solve It, Second Paperback Printing 1973, Princeton University Press, Princeton, N. J., 1973.

13. Koffman, E. B. "A Generative CAI Tutor for Computer Science Concepts," Proceedings fo the AFIPS 1972 Spring Joint Computer Conference, 1972.

14. Woods, W. A., "Transition Network Grammars for National Language Analysis," Communications of the ACM, Vol. 13, No. 10, October, 1974.

15. Winograd, T., Understanding National Language, Academic Press, New York, 1973.

16. Brown, J. S., Burton, R. R., and Bell, A. G., "SOPHIE" A Sophisticated Instructional Environment for Teaching Electronic Troubleshooting," Bolt, Beranek, and Newman Report No. 2790, Cambridge, Mass. March 1974.

APPENDIX A

EXAMPLES OF PROBLEM GENERATION

The following illustration uses nine basic problems from the area of elementary physics. Problem representations are generated at random. In order to reduce duplication generated abstract problems were hashed using the MACLISP system function SXHASH and the number values stored. Repetitions are still possible since, for example, (SXHASH '(A B)) ≠ (SXHASH '(B A)), yet (A B) = (B A) as sets. However, the amount of repetition is drastically reduced.

For this session the solution operators were executed to produce the solution routines. All of the generated routines have no lambda variables or prog variables; all the variables are global to the session.

Note problem representation 4. It was decided to allow the union of two problems where the input of one is the output of the other. The intent is that on an intepreted level this will take the form of two different instances of the same variable (e.g. an av-vel$_1$ and an av-vel$_2$).

Twenty-three distinct problems and their solution routines were generated before the session was terminated by typing in NIL. Problem 23 is a nice example. G0016 means that 16 'DEFUN' rotuines were generated.

Notice that the third basic problem was not output; the PLAN routine uses a random or nondeterministic control structure.

Complexity refers to the number of operators applied. It gives one measure of the difficulty of a problem and provides one way of defining a partial order.

```
(MAIN)
(ENTER OPERATOR NAME) PUNION
(ENTER OPERATOR NAME) PCOMPOS
(ENTER OPERATOR NAME) PCASCADE
(ENTER OPERATOR NAME) NIL
(ENTER PROBLEM TUPLES) ((FORCE)(MASS ACCEL) (SFORCE MASS ACCEL))
(ENTER PROBLEM TUPLES) ((ACCEL) (VEL1 VEL 2 TIME)(SACCEL VEL1 VEL2 TIME))
(ENTER PROBLEM TUPLES) ((AV-VEL) (VEL1 VEL2)(SAV-VEL VEL1 VEL2))
(ENTER PROBLEM TUPLES) (DIST)(AV-VEL TIME)(SDIST AV-VEL TIME))
(ENTER PROBLEM TUPLES ((ACCEL)(GRAV)(SEQ ACCEL GRAV))
(ENTER PROBLEM TUPLES)((WEIGHT)(MASS GRAV)(SWEIGHT MASS GRAV))
(ENTER PROBLEM TUPLES)((MOMENTUM)(MASS VEL)(SMOMENTUM MASS VEL))
(ENTER PROBLEM TUPLES)((IMPULSE)(FORCE TIME)(SIMPULSE FORCE TIME))
(ENTER PROBLEM TUPLES) NIL
(ENTER GO OR NIL) GO
(ENTER COMPLEXITY)-2
UNKNOWN:
(ACCEL)

DATA:
(GRAV)
```

```
SOLUTION:
(SEQ ACCEL GRAV)

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
UNKNOWN
2(CENTRI-ACCEL)

DATA:
(VEL RADIUS)

SOLUTION:
(SCENTRI-ACCEL VEL RADIUS)

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
UNKNOWN:
3(AV-VEL MOMENTUM)

DATA:
(VEL1 VEL2 MASS VEL)

SOLUTION:
(DEFUN G0001 NIL (PROG NIL (SAV-VEL VEL1 VEL2) (SMOMENTUM MASS VEL)))

(ENTER GO OR NILL) GO
(ENTER COMPLEXITY) 2
UNKNOWN:
4(DIST ACCEL AV-VEL)

DATA:
(TIME VEL1 VEL2)

SOLUTION:
(DEFUN G0002 NIL (PROG NIL (SDIST AV-VEL TIME) (SACCEL VEL1 VEL2 TIME)(SAV-VEL VEL1 VEL2)))

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
UNKNOWN:
5(ACCEL)

DATA:
(VEL1 VEL2 TIME)

SOLUTION:
(SACCEL VEL1 VEL2 TIME)

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2        duplicate problem generated
(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
GUNKNOWN:
(CENTRI-ACCEL FORCE)
```

```
DATA:
(VEL RADIUS GRAV MASS)

SOLUTION:
(DEFUN GOOO3 NIL (PROG NIL (SCENTRI-ACCEL VEL RADIUS) ( SFORCE MASS (SEQ ACCEL GRAV))))

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
ENTER TO OR NIL) GO
(ENTER COMPLEXITY) 2
UNKNOWN:
7(MOMENTUM IMPULSE)

DATA:
(MASS VEL FORCE TIME)

SOLUTION:
(DEFUN GOOO4 NIL (PROG NIL (SMOMENTUM MASS VEL) (SIMPULSE FORCE TIME)))

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
UNKNOWN:
8(ACCEL WEIGHT)

DATA:
(VEL1 VEL 2 TIME MASS GRAV)

SOLUTION:
(DEFUN GOOO5 NIL (PROG NIL (SACCEL VEL1 VEL2 TIME)  (SWEIGHT MASS GRAV)))

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
UNKNOWN:
9(DIST ACCEL)

DATA:
(AV-VEL VEL1 VEL2 TIME)

SOLUTION:
(DEFUN GOOO6 NIL (PROG NIL (SDIST AV-VEL TIME) (SACCEL VEL1 VEL2 TIME)))

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
UNKNOWN:
10(IMPULSE)

DATA:
(FORCE TIME)

SOLUTION:
(DEFUN GOOO7 NIL (PROG NIL (SIMPULSE FORCE TIME)))
```

```
(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
UNKNOWN:
11(ACCEL DIST)

DATA:
(GRAV AV-VEL TIME)

SOLUTION:
(DEFUN GOOO8 NIL (PROG NIL (SEQ ACCEL GRAV) (SDIST AV-VEL TIME)))

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 3
12UNKNOWN:
(CENTRI-ACCEL ACCEL WEIGHT)

DATA:
(VEL RADIUS MASS GRAV)

SOLUTION:
(DEFUN GOOO9 NIL (PROG NIL (SCENTRI-ACCEL VEL RADIUS) (SEQ ACCEL GRAV)(SWEIGHT MASS GRAV)))

(ENTER GO OR NILL) GO 3
(ENTER COMPLEXITY) 3
13 UNKNOWN:
(DIST FORCE CENTRI-ACCEL)

DATA:
(AV VEL TIME MASS ACCEL VEL RADIUS)

SOLUTION:
(DEFUN GOO10 NIL (PROG NIL (SDIST AV-VEL TIME) (SFORCE MASS ACCEL) (SCEN TRI-ACCEL VEL
RADIUS)))

(ENTER GO OR NIL)
(TRY AGAIN)
(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 3
14 UNKNOWN:
(WEIGHT CENTRI-ACCEL)

DATA:
(MASS GRAV VEL RADIUS)

SOLUTION:
(DEFUN GOO11 NIL (PROG NIL (SWEIGHT MASS GRAV) (SCENTRI-ACCEL VEL RADIUS)))
```

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 3
UNKNOWN:
15(FORCE)

DATA:
(MASS ACCEL)

SOLUTION:
(SFORCE MASS ACCEL)

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
UNKNOWN:
16(FORCE ACCEL IMPULSE)

DATA:
(MASS VEL1 VEL2 TIME)

SOLUTION:
(DEFUN GOO12 NIL (PROG NIL (SFORCE MASS ACCEL)(SACCEL VEL1 VEL2 TIME)(SIMPULSE FORCE TIME)))

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY)2
UNKNOWN:
17(IMPULSE ACCEL)

DATA:
(MASS VEL1 VEL2 TIME)

SOLUTION:
(DEFUN GOO13 NIL (PROG NIL (SIMPULSE (SFORCE MASS ACCEL) TIME) (SACCEL VEL1 VEL2 TIME)))

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
UNKNOWN:
18(DIST)

DATA:
(AV-VEL TIME)

SOLUTION:
(SDIST AV-VEL TIME)

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 1
(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 1
UNKNOWN:
19(MOMENTUM)

DATA:
(MASS VEL)

```
SOLUTION:
(SMOMENTUM MASS VEL)

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 1
UNKNOWN:
20(FORCE)

DATA:
(GRAV MASS)

SOLUTION:
(SFORCE MASS (SEQ ACCEL GRAV))

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 1
(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 1
(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
21UNKNOWN:
(IMPULSE FORCE)

DATA:
(VEL1 VEL2 TIME MASS)

SOLUTION:
(DEFUN GOOO12 NIL (PROG NIL (SIMPULSE FORCE TIME)(SFORCE MASS (SACCEL VEL1 VEL2 TIME))))

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY)2
(ENTER GO OR NIL) GO
ENTER COMPLEXITY) 2
UNKNOWN:
21(CENTRI-ACCEL WEIGHT)

DATA:
(VEL RADIUS MASS GRAV)

SOLUTION:
(DEFUN GOO15 NIL (PROG NIL (SCENTRI-ACCEL VEL RADIUS)(SWEIGHT MASS GRAV )))

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 2
(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 3
(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 3
UNKNOWN:
22(WEIGHT)
```

```
DATA:
(MASS GRAV)

SOLUTION:
(SWEIGHT MASS GRAV)

(ENTER GO OR NIL) GO
(ENTER COMPLEXITY) 3
UNKNOWN:
23(MOMENTUM IMPULSE)

DATA:
(VEL1 VEL2 MASS VEL TIME)

SOLUTION:
(DEFUN GO016 NIL (PROG NIL (SMOMENTUM MASS VEL)(SIMPULSE (SFORCE MASS (SACCEL VEL1
VEL2 TIME)) TIME)))

(ENTER GO OR NIL) NIL
GOOD-BYE
%
.K/T
26.45
Kilo-core-sec=860
```

APPENDIX B

ISLE'S BNF GRAMMAR

INPUT :: =<EDIT /<EVAL>/<REQUEST>

<EDIT>:: = any valid string of LISP edit commands

<EVAL>:: = any valid pair of items of the form function arglist for evalquote.

<REQUEST>:: = <DEFINE/Q>/<RELATION/Q>/ TEST/Q>/<COMPARISON/Q>/
<VAR/Q>/<#ARGS>/<FN/STRU/Q>/<EDIT/Q>/<FN/REF>

<DEFINE/Q>:: = { DEFINE / DESCRIBE / WHAT is/are* } <THINGS> [LIKE*]

*Does - DO ir DO - Do are also possible.

<COMPARISON/Q> :: = COMPARE <THINGS> [{ WITH / TO } < THINGS>]

WHAT IS/ARE THE DIFFERENCE(S) BETWEEN <THINGS>

[HOW] IS/ARE** <THINGS> { DIFFERENT FROM / SIMILAR TO / THE SAME AS } <THINGS>

[HOW] IS/ARE*** <THINGS> DIFFERENT***

** DO or DOES - DIFFER FROM also works.

*** SIMILAR, THE SAME, DO - DIFFER also work.

<RELATION/Q>:: = IS/ARE { IT, THEY / <FN/TYPE> / <FNNAME> / <STRUCTURES> } [KINDS of / A TYPE OF / etc.] { <FN/TYPE> / <STRUCTURES> }

<TEST/Q >:: = IS _____* { <STRUCTURES> / <FN/TYPE> / DEFINED }

* The underscore will match anything. In this way, sentences like: 'Is H123 on atom?' and "Is F3 defined?' will be recognized.

<VAR/Q>:: = WHAT [VARIABLE(S)] <FN & VRS> [THAT ARE <FN&VRS>]

<FN & VRS>:: = { DOES / CAN / WILL / etc } { IT / <FNNAME> } { BIND / SET / SET }

[ARE / CAN BE / MAY BE / etc.] { BOUND / SET / SET } { BY / IN } { <FNNAME> / IT }

<#ARGS >:: = HOW MANY ARGUMENTS* { FOR / DOES } { IT / FNNAME / FNTYPE } [HAVE / NEED / TAKE]

<FN/STRU/Q >:: = WHAT <FN/TYPE> { { ARE / IS } CALLED BY { IT, HIM, HER / <FNNAME> } [SOMEHOW / IN { SOME / ANY } WAY] ; { IT, HIM, HER / <FNNAME> } CALL(S) ; { DOES / CAN } { <FNNAME> / IT } CALL }

{ WHO / WHAT } { ---------- CALL(S)** { < FNNAME> / IT } ; { DOES / CAN / WILL / MAY } { <FNNAME> / IT, HE, SHE } CALL }

{ DOES / CAN } { <FNNAME> / IT } CALL [ANY / SOME] [OF] { <FNNAME> / <FN/TYPE> / IT, THEM, ITSELF }

{ IS / ARE } { <FNNAME> / IT } [USED / CALLED] [IN / FROM / BY] { <FNNAME> / <FN/TYPE> / IT / THEM }

* ARGS also works here.
** also possible are CAN, CALL, MAY CALL, etc.

Appendix B-3

<EDIT/Q>:: = { PRINT / EDIT / PRETTYPRINT } <FN NAME>

<FN/REF>:: = What is the { CAR / CDR / CADR / etc } of <STRUCTURES>

<THINGS>:: = { IT, THEY, THEM / <CONCEPTS> / <FN NAME> / <FN/TYPE> / <STRUCTURES> } [AND <THINGS>]